Simulation von Bedrohungen und deren Auswirkungen anhand eines Sicherheitsszenarios

Diplomarbeit

Marion Steiner

Prof. Dr. Claudia Eckert
Fachgebiet Sicherheit in der Informationstechnik
Fachbereich Informatik
Betreuer: Kpatcha Bayarou

10. Juni 2003



Inhaltsverzeichnis

Abbildungsverzeichnis

Erklärung

Erklärung zur Diplomarbeit gemäß §19 Abs. 6 DPO/AT

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 10.Juni 2003

1 Einleitung

Dezember 2000: Ein Darmstädter Großunternehmen, das nicht in der Computerbranche tätig ist, wird Opfer einer Denial-of-Service (DoS)-Attacke. Es wird zwar kein direkter Schaden angerichtet, aber durch zusätzliche Arbeitsstunden der Systemadministratoren entstehen Kosten von rund 10.000 DM. ([43])

Januar 2001: Eine DoS-Attacke gegen die Router von Microsoft schränkt das Webangebot des Softwareunternehmens besonders in den Bereichen microsoft.com und msn.com ein. ([44])

April 2002: Ein schottischer Internet Service Provider (ISP) wird durch einen DoS-Angriff beeinträchtigt. Teilweise werden zwei 45 Mbps-Leitungen komplett ausgenutzt, um Angriffspakete zu transportieren. Durch Installation eines Paketfilters kann der Angriff zwar beendet werden, aber erst nachdem einige Kunden des Anbieters bis zu 12 Stunden die ihnen zustehenden Dienste nicht nutzen konnten.[1]

Juli 2002: Die Webseite der „Recording Industry Association of America” wird ein ganzes Wochenende lang durch eine DoS-Attacke blockiert.[2]

Oktober 2002: 7 der 13 DNS Root Server sind eine Stunde lang durch einen DoS-Angriff unerreichbar. ([46])

November 2002: Die DNS-Server von UltraDNS, einem großen Anbieter für DNS-Services, werden 3 Stunden lang angegriffen. Obwohl sich hier 40 Rechner 2 IP-Adressen zum Lastausgleich teilen, erhält noch jeder Server mehr Pakete, als eine T1-Standleitung (ca 1.5 MBit/s) verkraften könnte. Der Angriff schafft zwar keine vollständige Auslastung der Systeme, dennoch beschließt UltraDNS eine Erhöhung ihrer Bandbreite.[3]

Diese Liste gibt nur einen kleinen Teil der bekannt gewordenen DoS-Aktivitäten wieder, die in den letzten Jahren stattgefunden haben[4]. Die Dunkelziffer ist sicher sehr hoch. Dennoch kann man daran sehen, dass Angreifer ihre Opfer nicht aus bestimmten Branchen wählen und dass sie Angriffe auf die zugrunde liegende Infrastruktur des Internets führen.

Noch immer verkennen viele Entscheidungsträger dieses Problem. Sie berufen sich darauf, dass niemand ein Interesse daran hätte, genau ihre Infrastruktur zu treffen. Viele Angriffe werden aber von Personen ausgeführt, die einfach nur testen wollen, ob ein Angriff funktioniert, wobei es ihnen ganz egal ist, wen sie

[1] siehe `http://www.theregister.co.uk/content/6/24773.html`
[2] siehe `http://zdnet.com.com/2100-1105-947101.html`
[3] siehe `http://www.theregister.co.uk/content/55/28291.html`
[4] Eine genauere Auflistung von DoS-Aktivitäten im Internet von 1999 bis 2001 ist [10] zu entnehmen. Weitere Beispiele in [42] und [45]

treffen. Die Schlussfolgerung „Keiner will mich absichtlich schädigen, daher bin ich in Sicherheit", ist unzutreffend. Weiterhin geschehen viele Angriffe auch aus Unachtsamkeit oder gar Unwissenheit der eigenen Mitarbeiter und könnten durch Aufklärung dieser User verhindert werden. Denn der Schaden, der durch diese Gruppe verursacht wird, ist nicht gering.

Deshalb ist eine Aufklärung über die bestehenden Gefahren notwendig!

Da statische Präsentationen zu diesem Thema nicht so wirkungsvoll sind und sich auch nur zu leicht als „das ist für mich aber ganz unzutreffend" abhandeln lassen, ist eine andere Art der Aufklärung notwendig. Am lehrreichsten wäre die Offenlegung der Sicherheitsmängel an der betreffenden Infrastruktur selbst. Aber dies ist wohl aus Kostengründen nicht möglich. Denn vor möglichem Schaden zu warnen, indem man die Produktionssysteme lahmlegt und den Schaden somit wirklich verursacht, ist kaum die richtige Methode. Die reale Infrastruktur zu duplizieren und dann auf dieser den Schaden zu zeigen, scheitert an der kostspieligen Anschaffung einer Menge an Hardware (und meist auch zusätzlicher Lizenzen für Software). Außerdem würden hierbei wahrscheinlich auch andere Internet-Teilnehmer geschädigt, oder zumindest behindert. Will man nämlich zeigen, dass z.B. eine Attacke auf das interne Firmennetz aus dem Internet möglich ist, so würde man die Angriffspakete auch durch das Internet verschicken müssen. Bei Angriffen, die auf großen Datenmengen basieren, wären also auch Router auf dem Weg der Angriffspakete und damit alle Internetnutzer, die auch Pakete über diese Router versenden, betroffen.

Daher stellt sich die Frage, ob man diese Gefahren nicht auf anderem Wege deutlich machen kann, beispielsweise durch Simulation. Durch eine Simulation könnte man die bestehende Infrastruktur kostengünstig abbilden und würde weder Produktionssysteme noch andere Internetnutzer stören.

Ziel dieser Arbeit ist deshalb, einen Prototyp einer solchen Simulation zu entwickeln und damit deren Machbarkeit zu zeigen. Da das Feld der möglichen Angriffe sehr weit gestreut ist, wird sich der hier entwickelte Prototyp mit der Simulation eines bestimmten Denial-of-Service-Angriffs beschäftigen, sowie mit einigen Maßnahmen, um sich gegen DoS-Angriffe zu wehren.

Die Arbeit ist wie folgend aufgebaut:

Kapitel 2 gibt einen Überblick über den Begriff des DoS („Denial of Service") und des „Distributed Denial of Service" (DDoS). Es werden Angriffstechniken erläutert und Probleme beschrieben, die von DoS ausgenutzt werden. Hier werden auch die Angriffe beschrieben, die durch den entwickelten Prototyp simuliert werden.

In Kapitel 3 folgt ein Überblick über mögliche Techniken zur Abwehr von DoS. Besonders wichtige Techniken werden genauer erläutert, um ihre Wirkungsweise während der Simulation nachvollziehen zu können.

Kapitel 4 zeigt die Grundlagen auf, die zur Simulation des Internets im Generellen und von DoS-Angriffen im Speziellen gebraucht werden.

Kapitel 5 beschreibt die Laufzeitumgebung des HLA-Tools, die für die Implementierung des Prototyps als Simulationsumgebung verwendet wird.

In Kapitel 6 werden die für die Simulation entwickelten Modelle erläutert. Diese beschreiben, wie sich ein Angriff auf den angegriffenen Rechner auswirkt, und wie sich die am Angriff Beteiligten verhalten.

Kapitel 7 entwickelt das Konzept, nach dem die Implementierung erfolgt ist.

In Kapitel 8 erfolgt eine Beschreibung der Implementierung des Prototyps. Neben Funktionsweisen der implementierten Klassen werden auch Designentscheidungen und konzeptuell bedingte Aspekte beschrieben, die bei der Implementierung berücksichtigt werden mussten.

Kapitel 9 zeigt dann die mit dem Prototyp durchgeführte Simulation von DoS-Attacken. Anhand dieses Simulationsbeispiels wird die Arbeitsweise des Prototyps deutlich gemacht. Hier erfolgt auch eine Anleitung, wie die dargestellten Meldungen der laufenden Simulation zu interpretieren sind.

In Kapitel 10 geben wir Hinweise, wie man den Prototyp weiterentwickeln könnte.

Kapitel 11 fasst die Ergebnisse der gesamten Arbeit noch einmal zusammen.

Anhang A erläutert, wie die Ausführung des Prototyps gestartet wird.

Anhang B beschreibt die im Prototyp verwendeten Klassen. Es werden die Funktionen und ihre Anwendung genauer erläutert.

2 Denial of Service (DoS)

Um einen Angriff simulieren zu können, muss man verstanden haben, wie der Angriff funktioniert. Man sollte dazu sowohl den zeitlichen Ablauf des Angriffs verstanden haben, als auch die Mechanismen, die im Lauf der Attacke verwendet werden. Ebenso ist es wichtig, sich das Ziel des Angriffs zu verinnerlichen, also die beabsichtigten Auswirkungen des Angriffs auf das Opfer.

2.1 Denial-of-Service-Angriff

Eine Denial-of-Service-Attacke (DoS-Attacke) hat zum Ziel, die Verfügbarkeit bestimmter Dienste eines Opfer-Servers zu stören. Dies wird erreicht, indem man z.B. den Opferrechner so mit falschen Anfragen auslastet, dass die ordentlichen Anfragen nicht mehr oder nur noch sehr unzureichend abgearbeitet werden können. Dabei werden Ressourcen des Opfers wie Speicher oder Prozessor(en) überlastet. Eine weitere Möglichkeit ist, dass man den Server gar zum Absturz bringt. Auch dann sind seine Dienste nicht mehr verfügbar.

Die Sicherheit der auf dem Server gelagerten Daten zu untergraben, sei es in Bezug auf Vertraulichkeit oder Integrität, ist nicht das Ziel von DoS-Angriffen.

2.2 Distributed Denial-of-Service (DDoS)

Das englische Wort „distributed” heißt im deutschen „verteilt”. Im Zusammenhang mit DoS bedeutet dies, dass man den Angriff auf das Opfer auf mehrere Rechner verteilt, die das Opfer attackieren. Beleuchten wir die Konsequenzen dieser „Verteilung” etwas genauer.

Im Unterschied zum normalen DoS-Angriff wird bei der DDoS-Attacke das Opfer nicht direkt vom Angreifer attackiert, sondern indirekt. Der Angreifer gibt den Rechnern, auf die er seinen Angriff verteilt hat, ein Zeichen, dass sie angreifen sollen, und diese führen dann je einen DoS-Angriff auf das Opfer aus. Hierzu benutzt der Angreifer ein Angriffsnetzwerk, über das der Angriff läuft. Dies dient einerseits dazu, die Rückverfolgung des Angriffs und damit die Entdeckung des Angreifers zu erschweren. Aber es gibt noch einen anderen wichtigen Grund: Durch dieses Netzwerk kann der Angreifer die Netzauslastung des Opfers erhöhen, ohne selbst eine hohe Bandbreite besitzen zu müssen.

Es gibt mittlerweile viele Möglichkeiten, solche Netzwerke zu etablieren. In dieser Arbeit wird allerdings mehr auf die klassischen Varianten der DDoS-Angriffe eingegangen. Zu diesen gehören fertige Angriffs-Tools wie „Trinoo”, „Tribe Flood

Network", „Stacheldraht" oder „Shaft" (siehe hierzu 2.6), die auch für Personen ohne spezielles Fachwissen anwendbar sind. DDoS-Attacken sind daher nicht auf Experten als Angreifer beschränkt.

2.3 Angriffspunkte für DoS-Attacken

Es gibt für den Angreifer drei Arten von Ansatzpunkten für seinen Angriff, mit denen er eine Auslastung des Opfers erreichen kann. Diese beruhen auf folgenden Problemen:

- **Programmfehler**
- **Protokollschwächen**
- **begrenzte Ressourcen**

Programmfehler sind Implementierungsfehler in Programmen, Protokollen oder Betriebssystemen. Meistens sind diese Fehler von der Art, dass Ausnahmen, die z.B. durch Fehleingaben von Programmen auftreten können, nicht abgefangen werden, oder dass das Programm nicht überprüft, ob behandelte Daten in den bereitgestellten Speicher passen („buffer overflow" oder auf deutsch „Speicherüberlauf").

Beispiele für Angriffe auf solche Programmfehler sind der „Ping-of-Death", bei dem das Empfangen eines manipulierten Ping-Pakets den Rechner zum Absturz bringt, weil das empfangene Paket mehr Speicherplatz beansprucht, als für das Paket bereitsteht.

Ein weiteres Beispiel ist der sogenannte OOB (Out of Bound) [5], bei dem es zu einem Crash kommt, wenn auf einem bestimmten Port andere Daten eintreffen als die erwarteten.

Bei **Protokollschwächen** ist das Problem nicht eine fehlerhafte Implementierung, sondern eine Designschwäche des Protokolls. Angriffe auf Protokollschwächen nutzen aus, dass manche Protokolle auf Serverseite (der Server ist hier auch das Opfer) deutlich mehr Ressourcen erfordern als beim Client (=Angreifer).[6]

[5] siehe hierzu [35], [25], [38]

[6] Zur Erklärung: ein Server ist ein Rechner, der einen Dienst anbietet. Ein Client ist ein Rechner, der einen Dienst bei einem Server anfordert. Da der Server oft Berechnungen anstellen oder große Datenmengen (z.B. Datenbankserver) bearbeiten muss, um die vom Client gewünschte Antwort bereitstellen zu können, ist der Aufwand einer Fragestellung des Clients oft deutlich geringer als der Arbeits- und damit Ressourcenaufwand, der auf der Seite des Servers entsteht.

Beispiele hierfür sind z.B. das 3-Wege-Handshake bei TCP, das nur eine begrenzte Zahl von gleichzeitigen Verbindungsaufbauversuchen erlaubt (siehe 2.4.2), oder CGI-Anfragen, die auf Serverseite deutlich mehr Prozessorkapazität erfordern als auf Clientseite, d.h. der Client kann mit wenig Arbeit eine große Auslastung des Servers erreichen.

Begrenzte Ressourcen ausnutzen heißt, man verwendet Dienste des Servers auf erlaubte Art, aber in einem Umfang, dass der Server die Flut von Anfragen nicht alle beantworten kann. Hierzu muss der Angreifer aber die selbe Bandbreite aufbringen, die er am Opfer belegen will. Solche Angriffe werden auch als Brute-Force-Angriffe oder Flooding-Attacken bezeichnet.

2.4 Beschreibung einiger wichtiger DoS-Attacken

2.4.1 Smurf

Smurf ist ein Angriff, bei dem das Opfer mit einer Flut von Paketen bedacht wird. Da diese Pakete in der Regel ICMP-Echo-Reply-Pakete sind, ist diese Attacke auch als ICMP-Storm bekannt. Den Namen Smurf erhielt sie nach dem ersten DoS-Tool, das sich den im Folgenden beschriebenen Effekt zum Angreifen zunutze machte.

Der Effekt, der bei einem Smurf-Angriff ausgenutzt wird, ist der, dass bei einem Ping auf die Broadcastadresse eines Netzwerkes alle Rechner im Netz dieses Ping-Paket (ICMP-Echo-Request) mit einem Echo-Reply-Paket beantworten. Dabei erhält man auf einfachem Wege eine Multiplizierung der Paketanzahl. Wenn man nun als Absendeadresse die Adresse des Opfers angibt, so kommen sämtliche Antworten beim Opfer an (Abb. 1). Das Angeben einer falschen Absendeadresse wird auch „spoofing" genannt.

Auch wenn sich das nicht so gravierend anhört, so wird ein einfaches Rechenbeispiel vom Gegenteil überzeugen. Wenn der Angreifer 1000 Echo-Pakete an eine Broadcastadresse eines großen Netzwerks schickt, bei der dann 100 Rechner antworten, so macht das dann 100000 Pakete, die das Opfer erhält. Das ganze in Bandbreite ausgedrückt: Sendet der Angreifer mit 128kb/s (DSL-Upload-Rate), so erhält das Opfer rechnerisch immerhin eine Auslastung von 12,8Mb/s (mögliche Download-Rate bei DSL nur 768kb/s).

An diesem Beispiel sieht man, dass man mit einem einfachen DSL-Anschluss schon einen anderen DSL-Teilnehmer auslasten kann, obwohl bei DSL mehr Bandbreite zum Empfangen als zum Senden bereitgestellt wird. Wenn nun aber der Angreifer vielleicht seinen Angriff von einem schlecht gesicherten Universitätsrechner startet, der über eine sehr hohe Bandbreite verfügt, so kann man problemlos auch Server lahmlegen, die über einen besseren Anschluss als DSL verfügen.

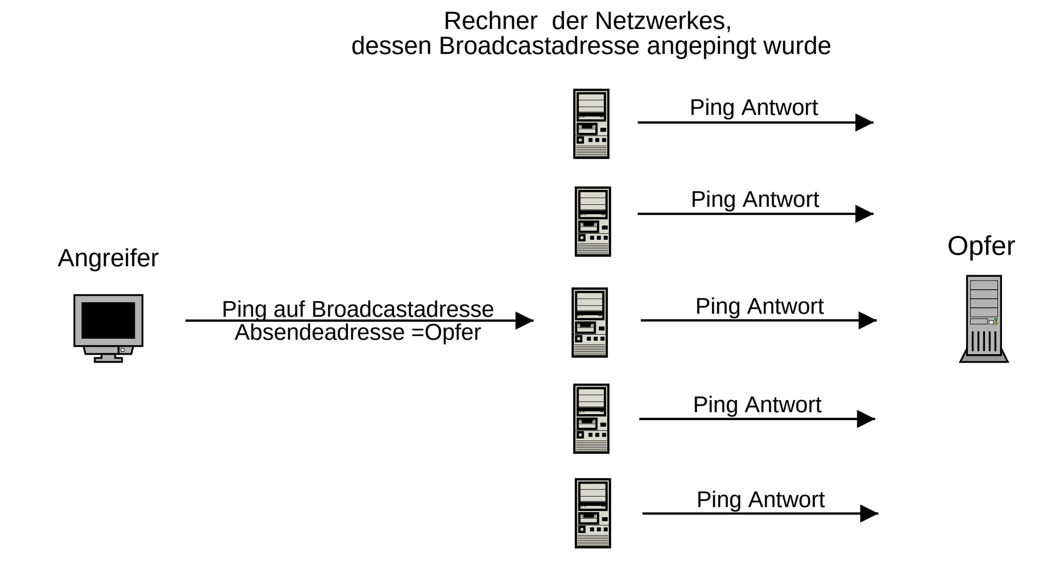


Abbildung 1: Smurf-Attacke

Weitere Informationen zu Smurf betreffend Vorgehensweise und Abwehrmaßnahmen sind in [21] zu finden.

Heise Online berichtete kürzlich von einer modernen Variante dieses Smurf-Angriffs, bei der Online-Gameserver für eine Attacke ähnlich der Smurf-Attacke verwendet werden ([47]). Diese Server bearbeiten über UDP normalerweise Informationsanfragen, wie z.B. die nach der Zahl der angemeldeten Spieler. Da sich bei UDP die Absendeadresse nicht überprüfen lässt, kann der Angreifer mittels Spoofing den Angriff auf sein Opfer lenken. Durch Kombination verschiedener Befehle an den Server kann der Angreifer einen Verstärkungseffekt auf etwa das 400-fache seines Aufwands bekommen. Somit genügt eine einfache 56Kb/s Modemverbindung um zwei T1-Standleitungen mit jeweils 1,56Mb/s auszulasten. Siehe hierzu auch [28].

2.4.2 SYN-Flooding

Das SYN-Flooding ist ein Angriff, der eine Schwäche im TCP-Protokoll ausnutzt. TCP ist ein verbindungsorientiertes Protokoll. Daher muss das Protokoll sicherstellen, dass alle Pakete der Kommunikation auch wirklich beim Partner ankommen. Aus diesem Grund bestätigen die Kommunikationspartner immer, welche Pakete sie erhalten haben. Hierzu ist eine spezielle Initialisierung der Verbindung nötig, das so genannte 3-Wege-Handshake (Abb. 2). Der hier beschriebene Angriff nutzt

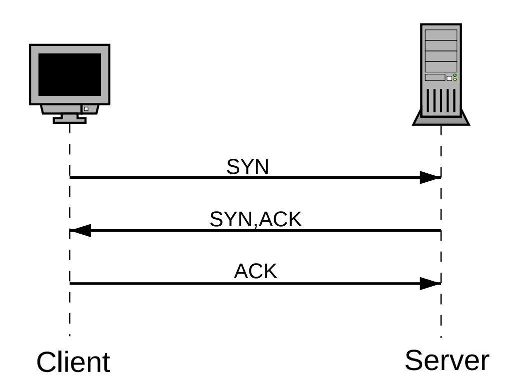


Abbildung 2: TCP-Verbindungsaufbau: Vollständiger 3-Wege-Handshake

genau dieses Handshake aus, um dem Server die Möglichkeit zu nehmen, weitere Verbindungen von außen einzugehen.

Das 3-Wege-Handshake funktioniert im Groben so: Der Client beginnt mit dem Verbindungsaufbau, indem er dem Server ein SYN-Paket sendet. Dieses wird dann vom Server mit einem SYN-ACK-Paket beantwortet. Die Verbindung ist aber erst dann hergestellt, wenn der Client das SYN-ACK noch mit einem ACK-Paket bestätigt. Da in einem Netzwerk Verzögerungen auftreten können, muss der Server sich jede durch ein SYN-Paket initialisierte Verbindungsanfrage solange merken, bis der Client den Verbindungsaufbau durch sein ACK-Paket bestätigt. Falls dies nicht geschieht oder zu lange dauert, wird der Verbindungsaufbau durch Erreichen eines TimeOuts abgebrochen.

Wenn nun also ein Angreifer einen Server mit TCP-Verbindungsanfragen überhäuft, dabei aber z.B. eine falsche Absendeadresse angibt, so kann kein Verbindungsaufbau erfolgen. Da das SYN-Paket gefälscht ist, sendet der Opfer-Server ein SYN-ACK an den Rechner, der die Adresse besitzt, von der das SYN-Paket angeblich kam. Dieser kann mit dem SYN-ACK aber nichts anfangen, da es zu keinem seiner Verbindungaufbauversuche gehört, und so ignoriert er das Paket. Das Opfer wird also vergeblich auf das ACK warten (Abb. 3).

Sendet der Angreifer nun sehr viele Anfragen, so erreicht der Server sein Maximum an gleichzeitig verarbeitbaren Verbindungsanfragen und kann keine neuen (auch legitimen) Verbindungen mehr annehmen, bis andere Verbindungen zustande kommen oder durch Erreichen des TimeOuts abgebrochen werden. Diese Möglichkeiten für neue Verbindungen können aber zum größten Teil vom Angreifer gleich wieder durch neue Anfragen belegt werden.

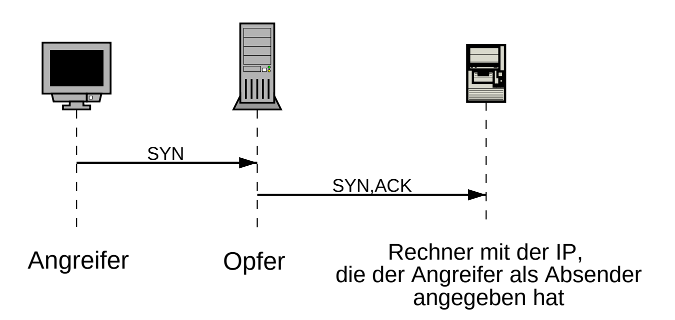


Abbildung 3: Connect-Versuch mit gespoofter Absendeadresse

2.5 Aufbau eines DDoS-Angriffs

In Abschnitt 2.2 wurde bereits erwähnt, dass der Angreifer bei einer DDoS-Attacke auf ein Angriffsnetzwerk zurückgreift. Er nutzt also viele fremde Rechner im Netzwerk, um eine Verteilung des Angriffs zu erreichen. Wie ein solches Netzwerk aussehen kann und wie der Angreifer es erzeugen kann, ist Thema der folgenden Abschnitte.

2.5.1 Angriffsnetzwerk

Es gibt viele Möglichkeiten, wie ein Angriffsnetzwerk bei einer DDoS-Attacke aufgebaut wird. Nun werden die zwei gebräuchlichsten Arten beschrieben, wie solche Netze aussehen können.

- **Einfaches Netzwerk:**

  Hier besteht das Netzwerk aus lauter gleichwertigen Komponenten, d.h Rechnern, die alle die selbe Aufgabe zu erfüllen haben. Diese greifen auf Initiative des Angreifers das Opfer-System an. Das kann entweder durch Senden eines Befehls an ein laufendes Programm auf den Netzwerkrechnern oder durch Ausnutzen von unsicheren Konfigurationen geschehen. Solche Netze stehen oft fertig bereit und können vom Angreifer mit minimalem Aufwand als Helfernetze missbraucht werden.

  Ein Angriff, der auf ein solches Netzwerk aufbaut, ist der so genannte Smurf-Angriff (2.4.1). Für den schematischen Aufbau dieses Netzwerks siehe Abbildung 4.

- **Master/Daemon-Netzwerk:**

  Dieses Netzwerk besteht aus zwei Typen von Rechner-Systemen: Master-Systeme und Daemon-Systeme. Ein Angriff erfolgt, indem der Angreifer den

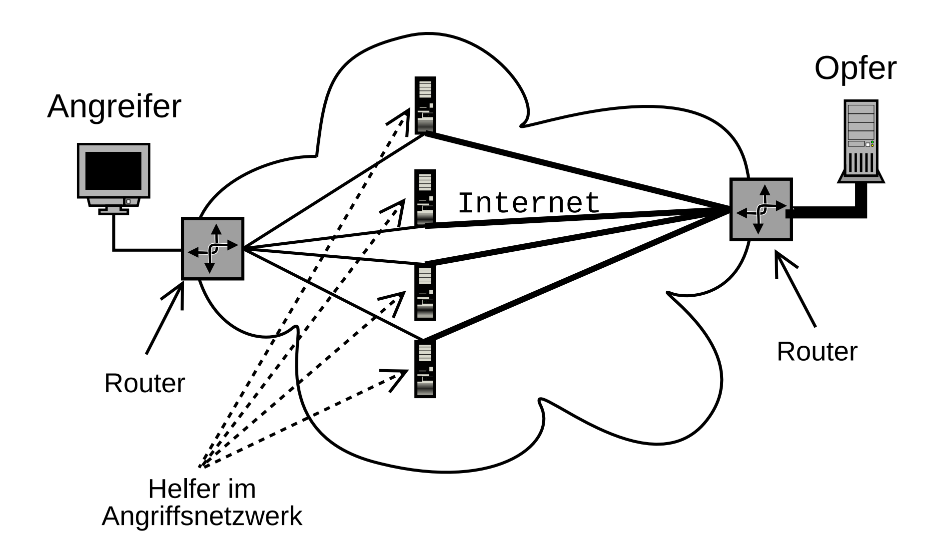


Abbildung 4: Beispiel: Einfaches Netzwerk

Master-Systemen den Angriffsbefehl erteilt. Diese senden daraufhin einen Angriffsbefehl an die Daemon-Systeme, die dann den eigentlichen Angriff auf das Opfer ausführen (siehe Abbildung 5).

Dieses Netz muss immer vom Angreifer aufgebaut werden, bevor er den Nutzen daraus ziehen kann.

Im weiteren Verlauf dieses Kapitels werden wir schwerpunktmäßig auf den Aufbau und die Vorgehensweise bei Master/Daemon-Netzwerken eingehen. Die Nutzung von einfachen Netzwerken für DoS-Attacken lässt sich daraus leicht ableiten.

2.5.2 Aufbau eines Master/Daemon-Angriffsnetzwerks

Wie schon erwähnt, muss der Angreifer sein Angriffsnetzwerk oft selbst aufbauen. Betrachten wir ein Master/Daemon-Netzwerk, so erfolgt der Netzaufbau im Wesentlichen in sieben Schritten:

1. Der Angreifer übernimmt auf einem Rechner mit vielen Usern einen Account, den er als Basis für die weitere Arbeit verwendet. Wie er diesen Account erhält, ob er ihn legal als angemeldeter User eröffnet oder ob er den Account eines Anderen dafür missbraucht, indem er sich fremde Passwörter erschleicht oder auf sonstigem Wege diesen Account knackt, sei hier nicht von Interesse.

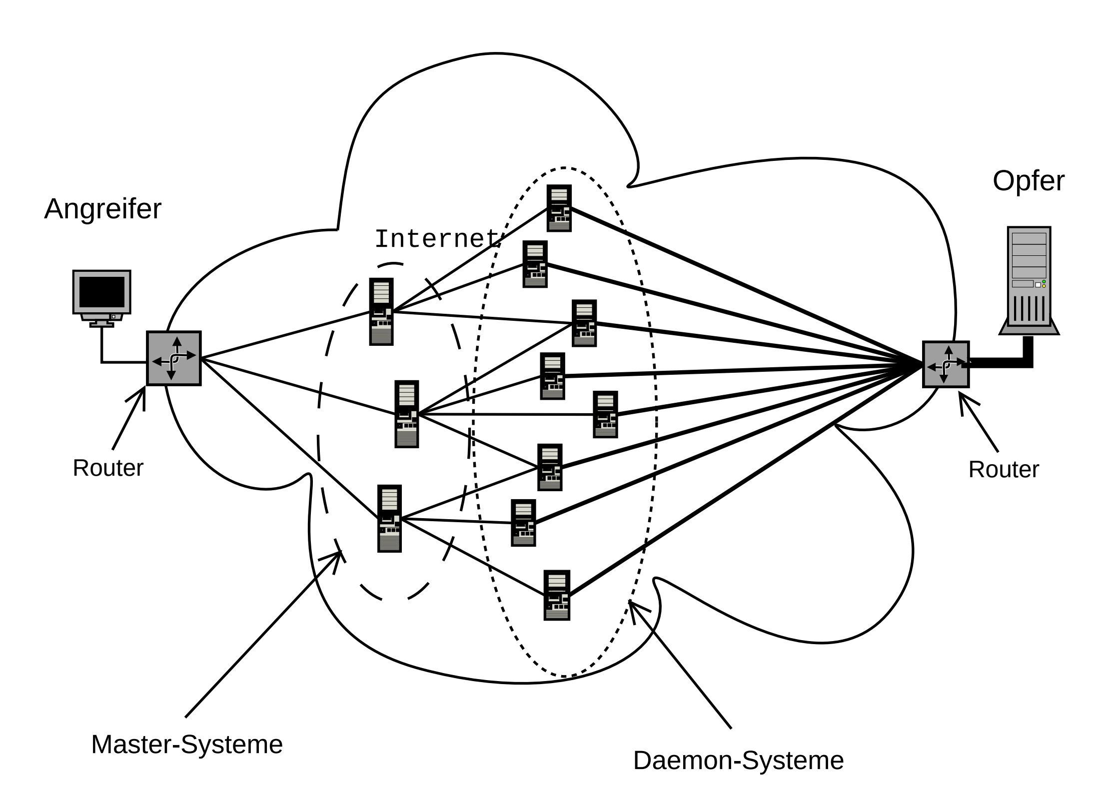


Abbildung 5: Beispiel: Master/Daemon-Netzwerk

Auf diesem Rechner lagert er die verschiedenen Tools, Skripte und Netzscanner, die er im weiteren Verlauf des Netzaufbaus braucht. Damit die Aktionen des Angreifers nicht auffallen, sollte dieser Rechner möglichst eine hohe Bandbreite besitzen. Die erhöhten Netzwerk-Aktivitäten eines Einzelnen fallen um so weniger auf, je mehr auch andere Personen von dem Rechner aus im Internet beschäftigt sind.

Ebenso könnte der Angreifer in vielen Fällen mit seinem eigenen Rechner, falls dieser WLAN-Möglichkeiten bietet, sehr viele offene WLAN-Netze finden, die ihm das Arbeiten im Internet ermöglichen. In diesem Fall wäre es nicht einmal nötig, sein Handeln besonders unauffällig zu gestalten, da der Angreifer bei Bedarf oder auch zum Selbstschutz sich jederzeit ein neues offenes WLAN-Netz als Operationsbasis suchen könnte.

2. Der Angreifer scannt das Internet auf Rechner mit bekannten Sicherheitslücken. Hierzu kann er z.B. ein Netzscantool verwenden, das er frei aus dem Internet bezieht.

3. Der Angreifer übernimmt Rechner, indem er die zuvor gefundenen Sicher-

heitslücken ausnutzt, um Root-Rechte auf den Systemen zu bekommen. Hierbei werden möglichst fertige Exploits[7] benutzt, um den Aufwand gering zu halten.

4. Der Angreifer teilt die übernommenen Rechner in Master- und Daemon-Systeme ein. Er nimmt die wichtigsten Rechner als Mastersysteme, die restlichen Rechner werden zu Daemons. Die Master-Systeme bilden nur einen geringen Anteil der gesamten Rechner des Angriffsnetzwerkes.

5. Die Installation der Daemon-Systeme erfolgt in der Regel automatisiert, d.h. skriptgesteuert. Mit „Installation" ist hier Folgendes gemeint: das Installieren der nötigen Programme auf dem System sowie das Verstecken der auffälligen Aktivitäten, die durch diese verursacht werden. Die Installation der Programme erfolgt meist durch Kompilieren des Quelltextes und Hinterlegen der ausführbaren Datei (meist bei Unix-Systemen) an einer bestimmten Stelle, oder durch Kopieren von fertigen Executables an den geeigneten Ort im Dateisystem. Das für die Installation verwendete Skript wird normalerweise als Teil der DoS-Tools bereitgestellt. Durch die Masse der Daemon-Systeme kann der Angreifer durchaus verkraften, nicht alle Daemons funktionsfähig zu bekommen. Das Fehlschlagen der Installation kann unter anderem an falschen Executables für ein System oder dem Fehlschlagen des Kompiliervorganges liegen.

6. Die Installation der Master-Systeme wird meist per Hand erledigt, da die Anzahl der Master-Systeme zu gering ist, um mehrere der Systeme nicht funktionstüchtig zu bekommen. Bei Installation per Hand kann der Angreifer besser auf das jeweilige System eingehen, Skripte und Programme so anpassen, dass sie beim Betrieb nicht auffallen, und vor allem auf Probleme bei der Kompilation der Programme eingehen. Um die installierten Programme und Hintertüren des Systems vor dem Entdecktwerden zu schützen, werden meistens Root-Kits verwendet, die u.a. bestimmte Systemfunktionen ersetzen und so die Ausführung bestimmter Anwendungen verstecken.

7. Nun kann der eigentliche Angriff erfolgen. Der Angreifer sendet den Angriffsbefehl mit der Adresse des Opfers an seine Master, diese geben ihn an die ihnen bekannten Daemon-Systeme weiter. Die Daemon-Rechner greifen nun die ihnen mitgeteilte Adresse, das Opfer, an.

[7] Ein Exploit ist ein Programm, welches speziell dafür geschrieben ist, Sicherheitslücken auszunutzen. Meist geschieht dies mit der Absicht, zu zeigen, was die Schwachstelle ist, und wie man sie ausnutzen kann.

Will man dieses Schema auf ein einfaches Angriffsnetz übertragen, so ist der Aufbau noch leichter: Da das einfache Netzwerk nur einen Typ von Helfern kennt, entfällt Aufzählungsschritt 4. Schritt 5 und 6 fallen zusammen.

Weiterhin sollte man sich aber auch bewusst sein, dass der Aufbau eines solchen einfachen Netzwerks oftmals gar nicht nötig ist, da viele im Internet erreichbare Rechner so konfiguriert sind, dass man sie schon als Helfer verwenden kann. Ein Beispiel hierfür ist wieder der Smurf-Angriff (2.4.1), da viele Router als Standardkonfiguration ICMP-Broadcasts im Netz weiterleiten und mancher Systemadministrator diese Einstellung nicht ändert.

2.5.3 Bemerkungen zu den Master-Systemen

Um den Angriffsbefehl weitergeben zu können, müssen die Master eine Liste mit IP-Adressen von Daemon-Systemen besitzen. Dabei braucht aber nicht jeder Master jeden Daemon zu kennen, sondern eben nur eine bestimmte Anzahl Daemons, für die er zuständig ist. Dabei können allerdings Daemons in den Listen mehrerer Master auftauchen, so dass immer noch (fast) alle Daemons erreichbar sind, selbst wenn ein Master ausfällt.

Um die Verwaltung einfacher zu gestalten, erhalten meist nicht die Master eine Liste von Daemons, für die sie zuständig sind, sondern die Daemons melden sich beim Aktivwerden bei den Mastern, zu denen sie gehören. Diese nehmen sie dann in die Liste der aktiven Daemons auf.

2.5.4 Bemerkungen zu den Daemon-Systemen

Daemon-Systeme brauchen keine Informationen über andere an dem Angriff beteiligte Systeme. Sie benötigen lediglich die Adresse des Opfers, das es anzugreifen gilt. Diese kann bei vielen Tools als Argument des Angriffsbefehls an den Daemon übermittelt werden.

Wenn allerdings wie in 2.5.3 beschrieben sich die Daemons bei ihren Mastern aktiv-melden, dann muss jeder Daemon eine Liste mit IP-Adressen von mindestens einem Master besitzen.

2.6 DDoS-Tools in der Praxis

Es gibt viele DDoS-Tools, die auf verschiedene Weise funktionieren. Um wirksame Gegenmaßnahmen entwickeln zu können, ist es nötig, die Tools zu analysieren und ihre Funktionsweise zu verstehen. So kann man unter anderem die Chancen erhöhen, die Tools nach ihrer Installation zu erkennen, indem man ihre Kommunikationssignaturen im Netzwerkverkehr aufspürt.

Es folgt deshalb eine kurze Einführung in die bekannten Tools. Hierbei wird vor allem Wert auf die Kommunikationseigenschaften gelegt. Die Aufstellung beginnt mit den Klassikern, die sich im Grunde alle sehr ähnlich sind. Im Laufe der Zeit wurden immer neue Tools geschrieben, die Verbesserungen gegenüber dem Vorgänger einführten, die besseren Schutz vor Entdeckung bringen sollen oder die Möglichkeiten des Tools erweitern.

Nach diesen klassischen Tools wird auf neuere Formen eingegangen, die nie einen solchen Bekanntheitsgrad erreicht haben wie die Klassiker. Hierzu werden vor allem die neuen Kommunikationsarten zwischen Mastern und Daemons erläutert, da sonst nicht viele Analysen vorhanden sind.

2.6.1 Klassiker

Die klassischen Tools arbeiten alle nach dem Schema, welches für das Master/Daemon-Netzwerk angegeben wurde (siehe Master/Daemon-Netzwerk in 2.5.1). Der Aufbau des für den Angriff verwendeten Netzwerks erfolgt analog zu dem in 2.5.2 beschriebenen Vorgehen.

Die Klassiker arbeiten alle mit direkter Kommunikation, d.h. der Angreifer redet direkt mit dem Master, und die Master kommunizieren direkt mit den Daemons.

Das Tool „Trinoo” ([11]) nutzt für die Kommunikation hohe UDP-Ports, über die unverschlüsselt gesendet wird. Als einzige Angriffsart kann es UDP-Flooding-Attacken ausführen. Die Daemons sind nicht in der Lage, ihre Absendeadresse zu fälschen, daher kann man Angriffe gut zurückverfolgen. Hat man die Daemons über ihre Absende-IP entdeckt, so kann man mit hoher Wahrscheinlichkeit das Daemon-Tool finden. In dieses sind die IP-Adressen einiger Master eincodiert, die man so auch aufspüren kann.

„Tribe Flood Network” (TFN, siehe hierzu [13]) bietet mehr Möglichkeiten als „Trinoo”. Als Angriffe sind neben UDP-Flooding auch TCP-SYN-Flooding, ICMP-Echo-Request-Attacken und ICMP-directed-Broadcast-Attacken (wie Smurf) durchführbar. Zur Kommunikation mit dem Master kann der Angreifer verschiedene Remote-Shells über die Protokolle UDP, TCP und ICMP benutzen.

Master und Daemons tauschen Informationen mittels ICMP-Echo-Reply-Paketen aus, damit ist diese Kommunikation sehr unauffällig. Die Weiterentwicklung von TFN zu TFN2k ([2]) brachte im Wesentlichen eine Verschlüsselung der Kommunikationsdaten mit sich.

Das „Stacheldraht”-Tool ([12]) führte gegenüber „TFN” neben Verschlüsselung der Kommunikation noch eine automatische Updatemöglichkeit der Daemons ein. Die Master benutzen nun eine TCP-Verbindung, um Daten an den Daemon zu übermitteln.

Weiter existieren die Tools „Shaft” ([14]) und „mstream” ([15]), die aber keine wirklichen Neuerungen mehr bringen. Vielmehr wurde mit jedem neuen Tool versucht, die Kommunikationssignaturen zu verändern, damit das Tool nicht so schnell entdeckt wird.

2.6.2 Neuere Methoden

Neuere Methoden verzichten auf die direkte Kommunikation. Damit IPs nicht im Quellcode festgelegt werden müssen, arbeiten diese Tools mit Vermittlungspunkten, z.B. IRC-Servern. So melden sich die auf den Helfern installierten DoS-Tools beim Aktivwerden nicht bei einem Master, sondern bei einem IRC-Server an. Über diesen erhalten sie auch die Befehle zum Angriff. Die einzige IP, die die Tools kennen müssen, ist also die des IRC-Servers.

Auf diese Weise wird bei Entdeckung eines DoS-Helfers nicht gleich die Aufdeckung großer Netzwerkteile ermöglicht. Ebenso ist die Kommunikation eines Rechners mit einem IRC-Server nichts Ungewöhnliches und normalerweise nicht schädlich. Daher wird die IRC-Kommunikation von den vielen Firewalls erlaubt. Das aktive Ausfiltern der Kommunikation von DoS-Tools wird damit sehr erschwert.

Eine Sammlung von Tools, von DDoS-Klassikern über IRC-Varianten, von Analysen der Tools bis hin zu Programmen zum Aufspüren installierter Tools ist unter ”`http://packetstormsecurity.nl/distributed/`” zu finden.

3 Maßnahmen gegen (D)DoS-Attacken

Um (D)DoS-Attacken zu verhindern, gibt es grundsätzlich verschiedene Ansatzpunkte, die man verfolgen kann. Man kann versuchen zu **verhindern, selbst das Opfer einer Attacke zu werden**. Das ist allerdings nicht so einfach, in manchen Fällen, wie z.B. bei Flooding-Attacken, sogar beinahe unmöglich. Weiterhin sollte man **versuchen, nicht als unfreiwilliger Helfer bei einer Attacke zu agieren**. Denn es ist einerseits für eine Firma nicht gut, wenn ihre Rechner andere Rechner angreifen, und andererseits könnte dieses Verhalten von anderen, auch nicht zum Helfer zu werden, einen Angriff auf die eigenen Server verhindern. Oder zumindest einen Angriff auf Server, von denen auch das eigene Netzwerk bzw. die eigene Außenkommunikation abhängt. Die letzte Möglichkeit, gegen DoS-Attacken vorzugehen, ist die **Traffic-Beobachtung im Netzwerk** selbst, die von den Betreibern der Router im Internet gemacht werden könnte. Diese können auffällige Datenströme im Netz gleich herausfiltern und so die Überlastung von benutzten Routern und damit auch manchen Angriff komplett verhindern. Hierbei stellt sich nur leider die Frage, warum die Routerbetreiber hierfür zusätzliche Kosten in Kauf nehmen sollten, wenn sie selbst davon nicht wirklich auch einen Vorteil haben.

Die nächsten Abschnitte befassen sich mit den vorhandenen Methoden zur Abwehr von DoS-Angriffen. Dabei werden zuerst Maßnahmen beschrieben, die auf einem Rechner zu seinem eigenen Schutz getroffen werden können. Danach folgt eine Beschreibung, wie man verhindern kann, als unfreiwilliger Helfer andere Rechner anzugreifen. Zum Abschluss des Kapitels werden Maßnahmen beschrieben, die auf Routern und Firewalls irgendwo im Internet oder auch im Heimatnetz des potentiellen Opfers getroffen werden können, um Angriffe gar nicht erst beim Opfer ankommen zu lassen.

Trotz dieser Aufteilung der Maßnahmen nach ihrer Rolle bei einem Angriff wird einem Systemadministrator die Anwendung aller ihm möglichen Methoden des Schutzes empfohlen. Es reicht nicht, sich nur gegen eine der Rollen abzusichern.

Für genauere Informationen, wie man sich gegen SYN- und Smurf-Angriffe wehren kann, wird für SYN-Abwehr auf [8] und für Smurf-Abwehr auf [28], [9] und [21] verwiesen.

3.1 Verhindern, Opfer zu werden

Will man verhindern, zum Opfer einer DoS-Attacke zu werden, so gibt es nur wenige wirksame Möglichkeiten. Denn Protokollschwächen lassen sich im Endeffekt nur durch Änderung des Protokolls selbst beheben. Füt das Handshake bei TCP

gibt es z.B. Methoden, die eine SYN-Flood-Attacke erschweren oder in vielen Fällen verhindern können. Aber im Prinzip sind diese tatsächlich Abänderungen des Protokolls, die dann das eigentliche Protokoll ersetzen, oder sie sind Wrapper, die fehleranfällige Teile zu verdecken versuchen. Die Außenwelt kommuniziert dann mit dem Wrapper, der Fehler abfängt und nur korrekte Angaben an das eigentliche Protokoll weitergibt. Meist haben diese Methoden aber auch einen Nachteil, sei es höherer Speicherverbrauch, schlechtere Performance oder gar die Möglichkeit, auch legitime Verbindungen zu behindern. Und diese Methoden müssen oft als Teil des Betriebssystems arbeiten, und damit vom Betriebssytemhersteller bereitgestellt werden. Ein Überblick zu Abwehrmaßnahmen und ihren Vor- und Nachteilen ist u.a. in [30] zu finden.

Zu den bekanntesten implementierten Methoden zur Abwehr von SYN-Attacken, die in einigen Standardbetriebssystemen (z.B. Linux) eingesetzt werden können, gehören „SYN-Cookies” oder das „random request dropping”. Eine kurze Erklärung zu diesen beiden Methoden ist [30] zu entnehmen.

Wie schon erwähnt, sollte man bei der Entwicklung von zukünftigen Protokollen darauf achten, diese sicher zu gestalten. Aber was heißt in diesem Zusammenhang sicher? Man kann versuchen, schon begangene Fehler zu vermeiden, aber vorauszusehen, was alles als Schwachstelle ausgenutzt werden kann, ist fast unmöglich. Und schließlich müssen wir heute auch die Protokolle verwenden, die schon existieren. Auf sichere Protokolle zu warten hat also keinen Sinn.

Die Theorie hat weiterhin Ansätze wie „Sicherheitsarchitekturen” entwickelt, die die Möglichkeit besitzen, Ressourcen von Verbindungen auf Ende-zu-Ende Basis zu verwalten und ihre Ressourcen zu begrenzen. Der Prototyp einer solchen Sicherheitsarchitektur schafft es, zuverlässig illegitime Verbindungen aufzuspüren und zu beenden und kann dabei Qualitätsgarantien einhalten. Leider aber ist er an ein Betriebssystem namens „Scout” gebunden und anscheinend nicht auf gängige Betriebssysteme zu übertragen ([30, 2.2.3]).

In der Praxis sollten Serverbetreiber auf keinen Fall vergessen, die offiziellen Patches gegen bekannt gewordene Schwachstellen in Betriebssystem und installierten Programmen einzuspielen. Denn keinesfalls sollte man die Gefahr, die von solchen Sicherheitslücken ausgeht, vernachlässigen. Für diese Lücken sind meistens kurz nach ihrem Bekanntwerden schon Exploits, also Programme zum Ausnutzen der Schwachstellen, im Internet zum Download zu finden. Und es gibt leider viele Skript-Kiddies, die nichts besseres zu tun haben, als diese Exploits aus dem Internet auszuprobieren und damit ohne jegliches Fachwissen Server zu überlasten. Der Aufwand, so gesamte Systeme zum Absturz zu bringen, ist gering. Aber herauszufinden, warum der Server abgestürzt ist, wenn manchmal schon ein einziges Paket dafür ausgereicht hat, dürfte schwer möglich sein. Deshalb ist es wichtig, solche

Updates möglichst schnell durchzuführen und sich auch über andere Schwachstellen zu informieren, für die es noch keine Fixes gibt. Der Administrator sollte bei solchen ungefixten Programmen gut überlegen, ob der Dienst nicht vielleicht ganz abgeschaltet werden kann, wenn er nicht so wichtig ist oder das selbe Angebot auf anderem Wege zur Verfügung gestellt werden kann.

Da viele Server nicht direkt im Internet hängen, sondern in kleineren Netzen, die durch einen Router vom übrigen Internet getrennt sind, hat der Serverbetreiber noch einige Möglichkeiten, Schutzmechanismen im Router einzusetzen. Die hierfür nötigen Einstellungen werden im Kapitel 3.3 erläutert, da viele dieser Einstellungen nicht nur für den letzten Router des Angriffspfades, sondern generell für jeden Router im Netz sinnvoll wären.

3.2 Schutz vor dem Helferwerden

Man muss sich nicht nur davor schützen, selbst Opfer einer DoS-Attacke zu werden. Es ist beinahe noch wichtiger, sich davor zu schützen, sich von einem Angreifer als Helfer rekrutieren zu lassen. Denn wenn der Angreifer es schafft, DoS-Tools auf einem fremden Server zu installieren, so hätte er meist auch anderen Schaden anrichten können: Daten verändern oder löschen, Programme zum Ausspionieren sensibler Daten installieren, etc. Deshalb ist auch hier das Einhalten bestimmter Regeln notwendig.

Wie schon beim Schutz vor dem Opferwerden ist auch hier das Einspielen von Sicherheitsupdates und Bugfixes zum schnellstmöglichen Zeitpunkt nach Erscheinen eine einfache Möglichkeit, das Risiko zu verkleinern. Und auch hier sollte man sich bei problembehafteten Diensten und Programmen ohne Fix entscheiden, ob man den Dienst nicht doch deaktivieren bzw. das Programm deinstallieren kann.

Während die gefährlichen Bugs für DoS-Opfer solche sind, die DoS Attacken begünstigen, sind die zur Helferrekrutierung verwendeten eher Buffer-Overflows [26], bei denen der Angreifer durch geschicktes Manipulieren des Stacks des Rechners die Möglichkeit erhält, Befehle auf dem Rechner auszuführen oder auf Daten zuzugreifen, auf die er sonst keinen Zugriff hätte. Neben den Sicherheitspatches, die solche Buffer-Overflows in betroffenen Implementierungen von Programmen und Protokollen beheben, gibt es aber noch Möglichkeiten, sich generell gegen die Auswirkungen von Buffer-Overflows zu wehren.

Um zu verhindern, dass der Angreifer die durch Manipulation auf dem Stack abgelegten Daten ausführen kann, kann man den Zugriff auf den Stack mit bestimmten Bits einschränken. Es gibt ein Bit für „Stack lesbar” und eines für „Stackcode darf ausgeführt werden”. Wenn nun die Bereiche des Stacks, in denen Daten und nicht

Programmteile liegen, als nicht ausführbar markiert werden, so kann man zwar nicht verhindern, dass durch einen Buffer-Overflow der Stack korrumpiert wird, aber die dort eingefügten Daten können ihre Schadfunktion nicht ausführen, da sie in nicht-ausführbaren Speicherbereichen liegen. Das Programm wird vielleicht abstürzen, aber es entsteht sonst kein Schaden.

Bei der im PC verwendeten x86-Architektur funktioniert dieser Ansatz nicht so einfach ([33, Background]). Hier muss das Betriebssystem Unterstützung leisten, damit die Nicht-Ausführbarkeit des Stacks unumgehbar wird. Implementierungen dafür gibt es unter anderem für Linux ([48]). Hierbei kann diese Funktionalität entweder über einen Patch dem Betriebssystemkern hinzugefügt werden ([33]), oder man muss den Quellcode von Programmen, die man verwenden möchte, mit speziellen Bibliotheken neu übersetzen. Dann erhalten die Programme selbst die Möglichkeit, die Schädlichkeit von Buffer-Overflows zu verhindern. Aber auch für Windows NT/2000 gibt es mit SecureStack von SecureWave einen entsprechenden Patch, der unter 2000 allerdings zu erheblichen Performance-Einbußen führen soll. Genaueres zu diesen Patches ist [26, Gegenmaßnahmen] zu entnehmen.

Aber auch fehlerfreie Dienste müssen auf sinnvolles Verhalten beschränkt werden. Bei den in 2.4.1 erwähnten Gameservern, die eine Verstärkerfunktion für Angreifer bereitstellen, könnte man z.B. maximale Bandbreiten festlegen, mit denen die Pakete an Ziele verschickt werden. Ebenso sollte man gerade bei diesen Gameservern eine Legitimation der Nutzer verlangen. Es könnten z.B. nur am Gameserver gerade angemeldeten Usern die Statusabfragen erlaubt werden. Wenn man für eine Anmeldung eine TCP-Verbindung verwendet, würde das den möglichen Opferbereich schon einschränken. Opfer könnten dann nur aus dem Userkreis des Gameservers ausgewählt werden.

Man sollte heutzutage auch daran denken, dass man sich durch Unachtsamkeit selbst zum Helfer machen kann. Sobald auf einem Rechner mit Internetverbindung aktive User erlaubt sind, die Mails lesen und Dateien und Programme aus dem Internet auf den Rechner laden, so sind diese User eine potentielle Gefahr für die Sicherheit des Rechners. Denn wenn ein Mailagent Viren verteilen und Dateien auf dem Rechner löschen oder verändern kann, so kann er auch DoS-Tools installieren. Oder statt dem virenbehafteten Bildschirmschoner oder dem 0190-Dialer lädt man sich freiwillig ohne es zu wissen das DoS-Tool eigenständig auf den Rechner.

Generell sollte man also Methoden verwenden, mit denen man (a) verhindern kann, dass unbeabsichtigt Programme installiert werden können, oder die (b) bewirken, dass neue Programme und Änderungen an Daten und Konfigurationen zumindest sofort bemerkt werden. Ein Verfahren für (a) ist die Verwendung von read-only Filesystemen. Programme zur Einbruchs-Erkennung (Intrusion Detection Tools (IDT)) versuchen an Punkt (b) zu greifen.

Bei read-only Filesystemen nutzt man den Effekt, dass ein potentieller Angreifer bei einem Einbruch in den Rechner keine Möglichkeit bekommt, Dateien abzuspeichern. Dadurch kann er weder neue Programme installieren, noch kann er bestehende Konfigurationen oder Daten manipulieren. Ganz so einfach ist das aber nicht, da man selten ein ganzes System nur auf read-only Filesystemen laufen lassen kann. Bei Systemen, auf denen User arbeiten, werden schreibbare Filesysteme gebraucht. Jeder andere Rechner, auch wenn er „nur" als Web-Server dient, braucht die Möglichkeit, variable Daten wie Prozessinformationen zu speichern. Wenn man aber zumindest die statischen Systemdaten, die sich bei normalem Arbeitsverhalten nicht ändern sollten, auf ein read-only Filesystem bringt, kann der Angreifer meist schon seine Tools nicht dort unterbringen, wo er sie haben möchte.

Wenn der Angreifer aber das System so kompromittiert hat, dass er Administratorrechte hat, kann er auch ein read-only Filesystem in ein schreibbares Filesystem umwandeln. Ein Dateisystem auf einer Festplatte also read-only zu mounten kann fast nur gegen unbeabsichtigte Installation durch Würmer, Viren oder automatisierte Tools helfen, die dieses nicht vorsehen. Um auch den auf dem Server selbst agierenden Angreifern Probleme zu machen, sollten die statischen Systemverzeichnisse möglichst auf read-only Speichermedien (z.B. CD-ROM) wirklich vor Veränderung geschützt werden.

Mittlerweile gibt es viele Intrusion-Detection-Tools, die auch sehr unterschiedliche Ansätze verwenden, um einen Einbruch in das System zu erkennen. Die einfacheren IDTs vergleichen beispielsweise nur, ob das Dateisystem verändert wurde. Dazu besitzen sie eine Datenbank, in der Informationen über statische Teile des Dateisystems enthalten sind. Wenn bei einer Überprüfung des aktuellen Dateisystems mit den Informationen aus der Datenbank Unstimmigkeiten gefunden werden, so wird Alarm ausgelöst. Um berechtigte Änderungen am System durchführen zu können, gibt es eine Updatefunktion für die Datenbank, die aber leider auch vom Angreifer benutzt werden kann. Die Datenbankdaten sollten also auch auf einem Medium gespeichert sein, welches Manipulationen durch Unberechtigte verhindert.

Modernere IDTs versuchen deshalb, nicht nur Dateistrukturen zu überwachen, sondern generell das Verhalten des Servers. Dabei würden auch ungewöhnlicher Netzwerkverkehr oder auffälliges Userverhalten bemerkt. Da sich User und Netzwerkverkehr aber nicht statisch verhalten, müssen diese Tools entweder einen statischen Bereich haben, in dem das Systemverhalten als „normal" gilt, und der bei Bedarf an neue Verhaltensweisen angepasst wird, oder sie haben eine Fähigkeit, neues Verhalten zu lernen. Dabei werden nur schnelle Veränderungen beim Verhalten bemerkt, schleichende Veränderungen würden als neuer „Normalzustand" gelernt. Damit könnte ein Angreifer aber auch dem System seinen Angriff als normales Verhalten beibringen. In beiden Fällen könnte aber auch ein User Fehlalarm

auslösen, da er sich trotz legaler Arbeit anders verhält, als das System es vorhersieht. Weitere Informationen können [6, Kapitel 12] entnommen werden.

Anstatt diese vielen Einzelmaßnahmen anzuwenden, gibt es auch Bestrebungen, Sicherheitsfunktionalität im Paket anzubieten. Hierzu zählt das Projekt Trusted-Debian-Linux[8], das viele Sicherheitsfunktionen bietet, unter anderem eingebaute Intrusion-Detection-Funktionalität und Sicherheit gegen Buffer-Overflows. ([49]).

3.3 Als Routerbetreiber DoS-Attacken verhindern

Ein wichtiger Teil der DoS-Bekämpfung kommt der Analyse und Filterung von Durchgangsverkehr in Routern zu. Hierbei gilt ein Prinzip: Man sollte versuchen, nicht legitimen Netztraffic so weit wie möglich an seiner Quelle abzufangen, damit jeder weitere Router auf der Strecke zum DoS-Opfer nicht unnötig belastet wird.

Der Weg eines IP-Pakets beginnt beim Absender. Dieser schickt das Paket, sofern es nicht in seinem Intranet lokal versendet wird, an einen Router. Dieser Router bildet in der Regel den Übergang des Intranets zum Internet. An diesem Übergang ist deshalb der Einsatz einer Firewall zu empfehlen.

Anhand der Beispiele der Smurf- (2.4.1) und der SYN-Attacke (2.4.2) ist zu erkennen, dass viele DoS-Angriffe mit der Technik des IP-Spoofings arbeiten, bei der die Absendeadresse des Opfers anstatt der des wirklichen Absenders im Paket angegeben wird. Eine Firewall, die den Ausgangsverkehr aus dem Intranet überwacht, sollte an dieser Stelle verhindern, dass offensichtlich gefälschte Absendeadressen das Intranet verlassen: Pakete, die nicht aus dem eigenen Netz stammen können, da ihre IP-Adresse nicht in das Netz passt, sollten von der Firewall verworfen werden, statt sie in das Internet weiterzuleiten. Diese Technik wird als „egress-filtering" (z.B. [22]) bezeichnet. Gefälschte Adressen, die innerhalb des Netzes gültig sind, kann man auf diese Weise nicht abfangen.

Während für diese Art der Filterung schon ein einfacher Paketfilter ausreichend ist, gibt es auch Filterkriterien, die höhere Anforderungen an eine Firewall stellen. So kann man dann auch Pakete mit beispielsweise einer ungültigen Kombination von TCP-Flags ausfiltern (Zustandsbasierte Filter) oder auch Inhalte auf Sinn oder Protokolle bestimmter Dienste auf Korrektheit untersuchen (application-level-gateway oder Proxies). Genauere Informationen zu den Möglichkeiten sind in gängiger aktueller Literatur zu Firewalls oder Netzwerksicherheit nachlesbar (z.B. auch [6, Kapitel 4], [16, Kapitel 11.1]).

Diese Techniken kann man auch auf das Zielnetz des Angriffs anwenden. Hierbei wird allerdings aus „egress-filtering" „ingress-filtering". Statt ausgehendem

[8] http://www.trusteddebian.org/

wird eingehender Verkehr gefiltert. Hier sollte also überlegt werden, welche Art von Paketen man in sein Netz hineinlassen möchte, welche internen Rechner erreichbar sein sollen, und welche Ports bzw Protokolle dort aus dem Internet verwendet werden dürfen. Ist im Intranet beispielsweise kein Web-Server, so braucht man keine Pakete mit Zielport 80 (Standardport für http-Dienste) in sein Netz zu lassen. Und hat man einen Web-Server, so reicht es im Allgemeinen, nur zu ihm Pakete mit Zielport 80 durchzustellen. Auch Pakete, die auf die Broadcastadresse gesendet werden, wie sie z.B. für Smurf-Angriffe genutzt werden, sind an diesem Punkt sinnvoll ausfilterbar. Denn wer außer Angreifern hat ein reelles Interesse daran, diese Adresse von außerhalb des Netzes zu verwenden?

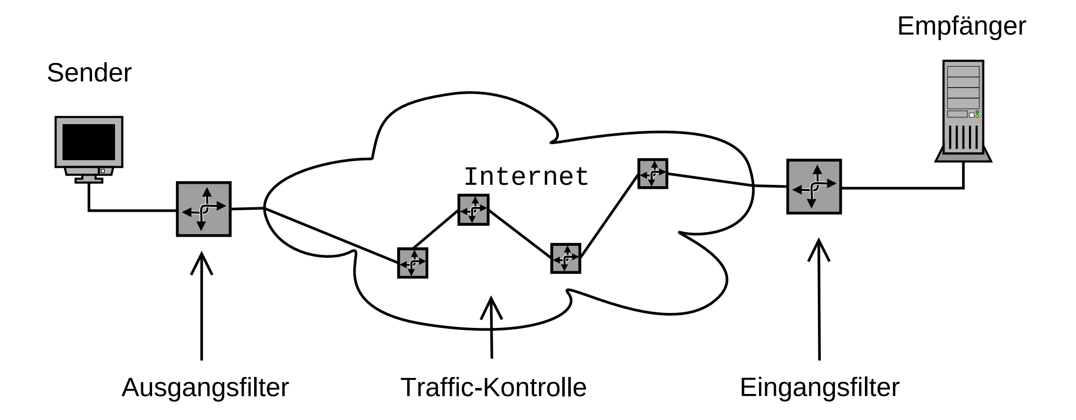


Abbildung 6: Stationen beim Paketversenden

Abbildung 6 zeigt beispielhaft den Weg eines Pakets vom Sender über verschiedene Router zum Empfänger. Hierbei wurde die „egress-filtering” betreibende Router-Firewall-Kombination als Ausgangsfilter bezeichnet, sowie die für „ingress-filtering” als Eingangsfilter. Die Router, die auf dem Weg des Pakets dazwischen liegen, können auch eine Kontrolle des Paketverkehrs übernehmen.

Dabei gibt es einmal den Ansatz, einfach Pakete, die zu einer Überlastung des Netzes führen, wegzuwerfen. Vereinfacht ausgedrückt meldet dabei ein Router, der überlastet ist, an alle Router, die ihm Pakete zusenden können, dass sie den Verkehr für ihn beschränken sollen, da er ihn nicht mehr komplett bearbeiten kann. Diese Router propagieren weiter, dass Pakete, die in Richtung des überlasteten Routers gesendet werden, nur in beschränktem Umfang an sie weitergeleitet werden. Damit werden auch legitime Pakete, und nicht nur Angriffspakete, ausgesondert, aber Paketverkehr, der ohnehin wegen Überlastung nicht bearbeitet werden könnte, wird schon möglichst früh ausgesondert, damit das Internet weniger belastet wird. Eine genauere Beschreibung dieses als „Push-Back” bekannten Verfahrens erfolgt in [37] und [24].

Andere Methoden versuchen, gezielt DoS-Pakete zu erkennen und zu verwerfen. Dabei wird der durchgehende Paketverkehr je nach Methode anders klassifiziert. Mal wird eine überdurchschnittlich große Paketmenge als Angriff bewertet, mal wird auf das proportionale Verhältnis von Hin- und Rückverkehr geachtet (siehe hierzu [19]), und mal wird ein Modell des normalen Traffics aufgestellt, dessen Verletzung zu Filteraktionen des Routers führt (siehe [32]).

4 DoS-Simulation

Simulationen versuchen, ein Abbild der realen Welt zu schaffen. Simulierte Handlungen sollen in diesem Abbild der Umwelt daher auch vergleichbare Reaktionen hervorrufen, wie sie in der Realität stattfinden würden.

Auf welchem Abstraktionsniveau diese Abbildung stattfindet, d.h. wie genau man Details der realen Welt nachbildet, ist abhängig von den Anforderungen an die einzelne Simulation. Man sollte die reale Welt nicht genauer abbilden als nötig, da man sonst unnötig komplizierte Modelle verwenden müsste. Andererseits kann man sich durch zu großes Abstrahieren sehr viele Möglichkeiten der Ausgestaltung nehmen. Es gilt also gut abzuwägen, inwiefern man Details berücksichtigt.

Warum eine solche Abbildung des realen Internets eine besonders schwere Aufgabe ist, und wie man ein geeignetes Abstraktionsniveau für eine DoS-Attacke ermittelt, wird im Folgenden erläutert. Zur weiteren Lektüre wird auf [18] verwiesen.

4.1 Simulation des Internets

Wenn man ein Abbild der Realität modelliert, entsteht meist ein Modell eines statischen Sachverhaltes. Damit kann man bestimmte Vorgänge in der Realität beschreiben, aber diese können kein Eigenleben entwickeln, sondern sich nur auf fest vorgeschriebenen Bahnen bewegen.

Das Internet ist aber eine sehr lebendige und komplexe Welt. Statische Aspekte gibt es in ihr kaum, sie besteht aus vielen verschiedenen Einheiten, deren Zusammenspiel erst das Internet zu dem macht, was es ist. Diese Einheiten reichen von den verwendeten Kabeln über die Protokolle, Anwendungen und Betriebssysteme, die von den Rechnern verwendet werden, um das Internet zu bilden, bis hin zu den Routern, die den Verkehr im Netz regeln.

Aber neben der Komplexität gibt es ein weiteres Problem. Das Internet ist riesig, und es entwickelt sich ständig weiter. Es wird zum einen größer, zum anderen ändert sich die Charakteristik der benutzten Protokolle. Während z.B. um 1992 der Schwerpunkt des Netztraffics bei FTP lag, fing das HTTP-Protokoll erst an, wirklich benutzt zu werden. Wenige Jahre später kehrte sich das Verhältnis von FTP zu HTTP um. Mittlerweile ist ein Hauptteil des Netzverkehrs auf Applikationen zur Multimediadatenübertragung zurückzuführen, zu denen unter anderem Napster oder Gnutella gehören. Vor wenigen Jahren war an solche Anwendungen noch nicht zu denken.

Aber nicht nur die Charakteristik der verwendeten Protokolle beeinflusst das Internet, sondern schon die Unterschiede in den einzelnen Implementierungen. Auch

diese Unterschiede können unter Umständen beim Simulationsverhalten zum Tragen kommen.

Wie aber soll man ein so komplexes, lebendiges Gebilde simulieren? Gibt es überhaupt eine sinnvolle Methode dafür?

Die einzig mögliche Antwort ist, dass man vereinfachen muss, indem man sich die Aspekte, die man für seine Simulation braucht, herausgreift und den Rest des Internets zu einem einheitlicheren, einfachen Modell zusammenfasst. Der größte Teil der Vitalität des realen Internets geht hierbei verloren. Das Wachsen des Netzes wird ignoriert. Man friert das Netzwerk sozusagen ein, macht ein Abbild zu einem bestimmten Zeitpunkt. Die Variabilität des Netzzustands beschränkt sich dann auf den Umfang, den der Entwickler dem Modell noch gestattet.

Kann man aber als weitere Vereinfachung die Größe des Netzes einfach verkleinern? Kann man das Internet sozusagen einfach als Netz mit beispielsweise 100 Rechnern abbilden?

Diese Frage lässt sich nicht generell mit „Ja" oder „Nein" beantworten. Es kommt auf Art und Ziel des Modells an und hängt von vielen Details wie z.B. den Eigenschaften der Protokolle ab. Manche Protokolle und Mechanismen funktionieren in Netzwerken bis zu einer bestimmten Größe wunderbar, sobald das Netz aber noch größer wird, arbeiten sie nicht mehr ordentlich. Will man solche Sachverhalte simulieren, dann kann man das Netz nicht beliebig verkleinert darstellen.

Bei der Simulation des Internets muss man sich eines weiteren Sachverhalts bewusst sein: Das heutige Internet ist nicht das Internet von Morgen. Das Netz verändert sich dazu zu schnell. Will man also neue Protokolle, Übertragungsarten etc. für das Internet testen, kann sich der Simulationslauf von der Realität dann doch unterscheiden, weil sich die Randbedingungen verändert haben. Man kann eben nur das Internet von heute oder früher simulieren, aber nicht das zukünftige. Für dieses kann man nur mögliche Entwicklungen zeigen, auf deren Eintreten aber kein Verlass ist.

Generell kann man sagen: Man muss sich zuerst darüber im Klaren sein, was man simulieren möchte, dann erst kann man eine geeignete Abbildung zu einem Modell finden. Denn die Anforderungen an das System bestimmen den Grad an Abstraktion, den ein Modell höchstens haben darf. Genauer werden darf man immer.

Betrachten wir als Beispiel eine Simulation, die die Performance verschiedener TCP-Implementierungen testen soll. Hier ist einsichtig, dass man das Protokoll 100% bis in die kleinsten Details nachbilden muss. Ebenso sind Anfälligkeiten gegen Netzverzögerungen, Paketverlust usw. zu berücksichtigen, die nötigen Details hierfür werden also auch gebraucht.

Bei einer DoS-Attacke hingegen kann man keine grundsätzliche Aussage über die nötige Abstraktion machen. Will man die Effizienz einzelner DoS-Tools vergleichen oder die Effektivität von Gegenmaßnahmen untersuchen, so ist eine detaillierte Modellierung unumgänglich.

Bei einer Simulation, die mehr Abläufe veranschaulichen als Protokolldetails beachten soll, ist eine solche Genauigkeit nicht angebracht. Es kommt nur darauf an, dass die auf Aktionen erwartete Reaktion eintrifft. Genaues Timingverhalten oder interne technische Abläufe können vernachlässigt werden. Das Bild, das die Modelle nach außen hin dem Betrachter zeigen, ist das Wichtige. Und nur dieses muss im Rahmen des Möglichen realistisch sein.

Die Simulation soll zeigen, wie sich Gegenmaßnahmen generell auf das Angriffsverhalten auswirken, und auch, wie in etwa solche Gegenmaßnahmen arbeiten. Zahlreiche Sonderfälle, die in wirklichen Netzwerken auftreten können, beispielsweise durch Race-Conditions, werden hierbei meist unberücksichtigt bleiben, sind aber auch im Rahmen einer solch visualisierenden Simulation nicht von entscheidender Bedeutung.

Wer Simulationsziele verfolgt, die genauere Darstellung von Protokollen und physikalischen Eigenschaften des Internets verlangen, sollte auf bestehende Netzwerksimulatoren zurückgreifen. Diese bieten oft schon Implementierungen der gängigen Protokolle an, so dass man sich auf die anderen Aspekte der Modellierung konzentrieren kann. Bekannte Simulatoren sind der Netzwerksimulator „ns”[9] oder das Simulationstool „x-Sim” (vgl. [5]). Diese bieten sogar die Möglichkeit, reale Hardware in die Simulation einzubeziehen. Man könnte also z.B. einen Angriff simulieren und ihn dann auf einen realen Server als Opfer anwenden. Oder man könnte ein echtes DoS-Tool benutzen, um einen Angriff in ein simuliertes Netz zu schicken. So werden gute Analysemöglichkeiten für Effizienz und Unterschiede der Tools geboten.

Für unsere Simulation jedoch ist ein solcher Netzwerksimulator eher ungeeignet.

Protokollverhalten und Effizienzbestimmungen von Implementierungen sind mit einem Netzwerksimulator gut möglich, aber schon die Interpretierung der Ergebnisse ist eine Kunst. Eine gut verständliche Präsentation, die auch von Personen ohne großes Fachwissen nachvollziehbar ist, ist damit unmöglich. Dazu werden zu viele Details berücksichtigt. Und einen Entscheidungsträger wird nicht interessieren, wieviele Daten nun genau bei einem Angriff ankommen, ihn wird nur interessieren: Sind wir noch funktionsfähig oder nicht, und wenn nein, wie können wir etwas dagegen tun.

[9] Informationen zu ns auf `http://www.isi.edu/nsnam/ns/`

Aus diesen Gründen werden wir einen eigenen Simulator entwickeln, der besonders die für Präsentationen wichtigen Aspekte des Netzgeschehens aufzeigen soll. Eine Vergleichbarkeit der DoS-Tools wird er aber nicht bieten.

4.2 Spielwiese: DoS-Attacken

DoS-Attacken sind weit verbreitet, und man kann an ihnen viele Verhaltensweisen aufzeigen, die zur Verstärkung der Probleme oder auch zu ihrer Abschwächung beitragen. Deshalb wollen wir uns bei unserer Prototyp-Simulation auf (D)DoS-Attacken beschränken.

Unbedarfte Internetuser handeln z.B. oft nach dem Motto „Wenn mich jemand angreift, dann schlage ich zurück”. Da kann schon ein vermeintlicher Portscan des heimischen Rechners mit einem massiven DoS-Angriff beantwortet werden. Beliebt ist hier unter anderem der Versuch, den Angreifer dazu zu bringen, über sein Modem den String '+++ATH0' zu senden, das bei vielen Modems zu einem Verbindungsabbruch führt, weil sie es als Kommando zum Trennen der Verbindung interpretieren[10].

Aber es könnten auch andere Reaktionen denkbar sein, die auch Nicht-Modem-Nutzer beeinträchtigen können. Denkt man, dass eine Reaktion nicht auf einen Portscan erfolgt, sondern auf einen richtigen Angriff, so könnte der Angegriffene leicht zu Gegen-DoS-Angriffen als „Abwehrmaßnahme” greifen wollen. Ist nun aber jeder dieser Angriffe mit falscher Absendeadresse versehen, so könnte ein DoS-Angriff zu einer lawinenartigen Ausbreitung weiterer Angriffe führen. Den meisten Internet-Nutzern ist dies nicht bewusst.

Neben den verstärkenden Effekten kann man gut die Auswirkung von Schutzmaßnahmen (Kapitel 3) zeigen. Hierbei kann man von bisher rein theoretischen Ansätzen bis hin zu häufig verwendeten Methoden vieles zeigen.

Für die Aufklärung der Entscheidungsträger in Firmen sind besonders praktisch anwendbare Methoden interessant. Ihnen könnte man die Funktion einer Firewall veranschaulichen. Vielleicht würde dann in einigen Firmen doch die sinnvolle Konfiguration dieser Firewall und der Schutz vor E-Mail-Schädlingen mehr Unterstützung erhalten, als das Verlangen der User nach Anwenderfreundlichkeit, die oftmals unbeabsichtigt auch Virenfreundlichkeit bedeutet.

Wenn man den Verantwortlichen die Potentiale der Schutzfunktionen zeigen kann, und daneben auch die Risiken, die ein potentieller Angriff darstellt, verbunden mit den auftretenden Kosten, so kann dies bei der Entscheidung für oder gegen

[10]vgl. http://www.opennet.ru/base/usersoft/3.txt.html

bestimmte Maßnahmen helfen. Man lernt, die Funktionen besser einzustufen und kann so das Kosten/Nutzen-Verhältnis besser einschätzen. So werden bestimmt einige Maßnahmen trotz zusätzlicher Kosten als lohnend erkannt werden.

In unserem Prototyp wollen wir uns deshalb vorerst mit praktischen Abwehrmethoden wie Paketfilter-Firewalls befassen. Ebenso sollen ein einfaches Intrusion-Detektion-Tool entwickelt und die Möglichkeit von Read-only-Filesystemen vorgestellt werden. Diese Verfahren werden hier in einer einfachen Variante vorgestellt. Modernere praktische Anwendungen haben mehr Moglichkeiten als die simulierten. Aber es geht hier zunächst um einen „Proof of Concept”, der die Machbarkeit darstellen soll, und nicht ihre sämtlichen zukünftigen Möglichkeiten.

5 Simulationsumgebung

Eine Simulation kann aus mehreren voneinander unabhängig arbeitenden Komponenten bestehen. Jede der Komponenten ist schon eine eigenständige Simulation, aber die gewünschte Gesamtsimulation besteht aus vielen dieser kleinen Simulationen. Das hat den Vorteil, dass die Simulation einfacher zu variieren ist. Da jede Komponente unabhängig von den anderen ist, können Teilkomponenten gegeneinander ausgetauscht werden, ohne Veränderungen des Programmcodes nötig zu machen.

Eine solche Simulation nennt man „verteilte Simulation", wenn die Komponenten auch noch auf unterschiedlichen Rechnern ausgeführt werden.

Bei einer Simulation des Fluges eines Flugzeugs könnte man beispielsweise eine Komponente *Flugzeug* haben, die das Flugzeug simuliert. Eine andere wäre für die Erzeugung der Umweltdaten wie Windgeschwindigkeiten, Wetterlage etc. zuständig. Wir nennen sie hier *Wetter*. Um nun z.B. die Variation der Flugzeit in Abhängigkeit von der Wetterlage zu bestimmen, braucht man den Quelltext *Flugzeug* nicht zu verändern. *Flugzeug* erhält nur die Angaben über die Umwelt, die *Wetter* ihm liefert. Man kann dann einen Wettererzeuger *Wetter2* schreiben, der alternative Wetterdaten liefert. Dann kann man diese Wetterdaten auch für einen zweiten Flugzeugtyp *Flugzeug2* verwenden, um für ihn auch die unterschiedlichen Flugdaten zu ermitteln.

Damit diese Komponenten sinnvoll kooperieren können, braucht man einige Hilfsmittel. Es wird eine gemeinsame Zeitbasis benötigt, die Simulationsteile müssen miteinander kommunizieren und Daten austauschen können.

Unser Prototyp wird für diese Funktionen auf die „High Level Architecture" (HLA) zurückgreifen, die die angesprochenen Hilfsmittel bereitstellt. HLA bietet noch viel mehr, als wir hier verwenden werden. Deshalb werden wir uns in der folgenden Beschreibung auf die wichtigsten Aspekte beschränken. Für eine genauere Beschreibung von HLA wird auf [29] verwiesen.

Geklärt werden zunächst die Begriffe, die für den Umgang mit der HLA wichtig sind. Danach werden die nötigen Mechanismen beschrieben, sowie kurz die Funktionen erklärt, die in unserem Prototyp Verwendung finden.

5.1 Begriffe

Die Gesamtsimulation, deren Teile über die Laufzeitumgebung miteinander kooperieren, nennt man **Federation**.

Die einzelnen Teile, die zu einer Federation zusammengesetzt werden, werden als **Federate** bezeichnet. Diese Federates können eigenständige oder auch verteilte Simulationen sein, sie können aber auch Interfaces zu Real-world Komponenten oder passiven Datensammlern oder Visualisierungskomponenten sein.

Ein Simulationslauf, bei dem eine Federation zur Simulation ausgeführt wird, ist eine **Federation-Execution**.

Zu einem Simulationslauf braucht man neben einem Satz von Federates die **RTI** (RunTime Infrastructure = Laufzeitumgebung), deren Spezifikation durch HLA gegeben wird. Die RTI ist implementierungsabhängig, d.h. einzelne Details wie Aufrufparameter, genaue Benennung von Funktionen, etc. können sich von Implementierung zu Implementierung unterscheiden. Hierzu sei ein Blick in die Referenz der verwendeten Implementierung empfohlen (hier RTI-NG 1.3 v6 des DMSO (Defense Modeling and Simulation Office)).

Damit die RTI die Kommunikation der Federates verwalten kann, muss ein „Federation Object Model” (**FOM**) erstellt werden. Mit diesem werden die Daten und Events festgelegt, die die Federates zum Datenaustausch verwenden können. Dieses FOM ist vom Entwickler der Simulation zu erstellen.

5.2 Federation Management

Damit RTI und Federate kommunizieren können, brauchen beide je ein Interface, das vom Anderen angesprochen werden kann (Abb. 7). Dazu stellt die RTI einen RTI-Ambassador zur Verfügung, an den der Federate seine Meldungen (Events) schicken kann. Umgekehrt muss der Federate einen Federate-Ambassador implementieren, damit er die Events der RTI entgegennehmen kann. Das Interface für den Federate-Ambassador ist durch HLA spezifiziert worden.

Für jede Simulation wird eine Federation-Execution benötigt. Daher gibt es eine RTI-Funktion, mit der eine solche erzeugt werden kann: *createFederationExecution()*. Bei der Erzeugung der Federation-Execution werden die Angaben über den Namen der Execution und das `.fed`-File (fed=*f*ederation-*e*xecution *d*ata) benötigt, in dem das FOM der Execution definiert ist. Nach Ende der Simulation kann die Execution mit der Funktion *deleteFederationExecution()* wieder beseitigt werden. Sie wird nicht mehr gebraucht.

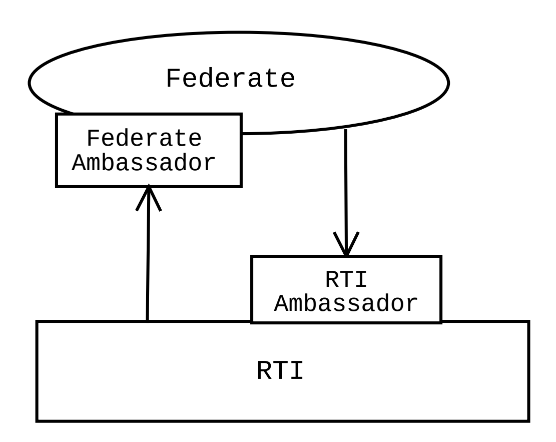


Abbildung 7: Kommunikation zwischen Federate und RTI

Jeder Federate muss einer Federation angehören, damit er Events über die RTI verschicken und erhalten kann. Denn mit dieser Federation sind das FOM und die anderen Teilnehmer der Simulation festgelegt. Das Beitreten zu einer Execution erfolgt über den Befehl *joinFederationExecution()*. Hierbei muss der Federate der RTI seinen Federate-Ambassador bekanntmachen, damit die RTI diesen verwenden kann.

Der Federate kann nach Beendigung seiner Aktivitäten die Federation-Execution mit dem Aufruf der RTI-Funktion *resignFederationExecution()* verlassen.

HLA bietet auch Funktionen, um eine Federation-Execution zu synchronisieren. Dazu kann man in den Federates **Synchronisationspunkte** setzten, an denen der Federate solange wartet, bis alle anderen Federates auch diesen Punkt erreicht haben. Dies ist z.B. sinnvoll, um bei einer Simulation erst alle Federates ihre Initialisierung beenden zu lassen, bevor das eigentliche Simulieren beginnt. Man würde deshalb am Ende der Initialisierungsphase jedes Federates einen Synchronisationspunkt setzen. Erst wenn alle Federates diesen Punkt erreicht haben, also wenn jeder Federate fertig initialisiert ist, dürfen sie weiterarbeiten.

5.3 Zeitverwaltung

Wollen mehrere Komponenten an einer Simulation teilnehmen, so sollten sie sich auf eine gemeinsame Zeitbasis einigen. Bei der verteilten Simulation von Abläufen ist das notwendig, um Abläufe zwischen den verschiedenen Teilen in korrekter Reihenfolge darstellen zu können. Denn wenn die Einzelkomponenten auf unterschiedlichen Rechnern ausgeführt werden, werden die Uhren dieser Rechner unterschiedlich laufen. Weiter müssen Verzögerungen beim Datenaustausch über das

Netzwerk berücksichtigt werden, die sich zeitlich nicht genau vorherbestimmen lassen. Daher bietet HLA eine solche gemeinsame Zeitbasis.

Die HLA bietet den Federates verschiedene Möglichkeiten, das HLA-Zeitmanagement zu verwenden. Unter anderem kann jeder Federate bei ihr die Optionen *timeRegulation* und *timeConstrained* setzen. Diese sagen der RTI, wie sie den Federate in das Zeitmanagement einfügen müssen. Ein Federate, der die Option *timeConstrained* setzt, wird von der RTI nur Meldungen bekommen, die eine fortgeschrittenere Simulationszeit besitzen, als er selbst. Mit der Option *timeRegulation* verpflichtet sich der Federate, keine Meldungen an die RTI zu schicken, die aus seiner Vergangenheit stammen. Seine frühsten Meldungen dürfen einen Zeitstempel besitzen, der sich aus $t_{fed} + LookAhead$ berechnet, wobei t_{fed} die aktuelle Zeit des Federates ist und $LookAhead$ ein konstanter positiver Wert.

Dieser LookAhead dient dazu, dass jeder Federate nur Meldungen erzeugen kann, die einen Zeitstempel besitzen, der in ihrer Zukunft liegt. Den Sonderfall, dass dieser LookAhead den Wert 0 annehmen darf, vernachlässigen wir hier. Wir verweisen dazu auf [29], für uns ist dieser Fall nicht relevant.

Die gesetzten Zeit-Optionen werden von der RTI berücksichtigt, wenn ein Federate in der Zeit voranschreiten will. Der Federate ruft hierzu die **Time-Advance**-Funktion der RTI auf, er meldet damit, dass er die Arbeit seiner aktuellen Zeit beendet hat und weitergehen möchte. Die RTI muss ihm dazu die Erlaubnis geben. So kann sie für die Federates auf die Einhaltung der gesetzten Zeitoptionen achten. Sie lässt einen Federate, der die Option *timeConstrained* gesetzt hat, nicht zur Zeit t_n voranschreiten, bevor nicht der letzte Federate mit der Option *timeRegulation* zu dieser Zeit vorangeschritten ist. Dies ist eine etwas vereinfacht Darstellung, denn eigentlich muss an dieser Stelle noch der LookAhead berücksichtigt werden. Zum Verständnis der HLA für unsere Simulationszwecke reicht dieses vereinfachte Modell aber aus.

Die Erlaubnis zum Time-Advance gibt die RTI dem Federate über die Federate-Ambassador-Funktion *timeAdvanceGrant()*.

Der Federate erhält von der RTI Events nur in der Zeit, in der er auf den *timeAdvanceGrant* wartet. Dieses Verhalten kann mit der Funktion *enableAsynchronousDelivery()* verändert werden. Dies ist in unserem Prototyp aber nicht der Fall.

5.4 Datenaustausch zwischen den Federates

Die Kommunikation der Federates soll ausschließlich über die RTI laufen, denn sonst könnten die HLA-Mechanismen für Time-Management und Synchronisation ausgehebelt werden. Auch die mögliche Verteilung der Simulation würde gefährdet.

Daher bietet HLA zwei verschiedene Möglichkeiten des Datenaustauschs: Über *Interaktionen* und über *Objekte*.

Der Datenaustausch erfolgt nach dem Erzeuger-Verbraucher-Prinzip: Es gibt eine Federate-Gruppe, die Daten erzeugt, und es gibt eine Gruppe, die sie konsumiert. Dazu gibt es RTI-Befehle, mit denen man seine Absicht kundtun kann, Daten zu veröffentlichen bzw. zu erhalten, falls jemand sie veröffentlicht hat.

Objekte sind dabei dauerhaft angelegte Objekte, bei denen die Federates einzelne Attribute verändern können und dann auch nur über die Veränderungen der Attribute informiert werden, die sie interessieren. Es darf immer nur ein Federate zu jeder Zeit die Erlaubnis besitzen, ein Attribut zu verändern oder gar das gesamte Objekt zu löschen. Für unseren Prototyp benutzen wir keine Objekte, deshalb erklären wir sie hier nicht weiter.

Interaktionen sind immer eine Einheit. Entweder möchte ein Federate eine Interaktion erhalten oder nicht. Dabei kann auch eine Interaktion mehrere Parameter besitzen. Mit der RTI-Funktion *receiveInteraction()* informiert man die RTI über die Interaktions-Klassen, die man von ihr gemeldet bekommen möchte, mit *publishInteraction()* gibt man die Interaktions-Klassen bekannt, die man selbst erzeugt.

Alle *Interaktionen* und *Objekte*, die in einer Simulation verwendet werden können, müssen durch das FOM definiert werden. Die hierzu definierten Klassen müssen dabei immer von der Klasse *InteractionRoot* (als Interaktion) bzw. *ObjectRoot* (als Objekt) oder einer Unterklasse dieser Root-Klassen abstammen.

In Abbildung 8 ist die Definition der Interaktionen des Prototyps dargestellt. Hierbei gibt es einmal die für die Simulation speziell entwickelten Interaktionen, aber auch schon von der RTI definierte. Diese sind nötig, damit die RTI auch andere Events, wie beispielsweise die Bekanntgabe des *timeAdvance*, an den Federate übermitteln kann.

Obwohl die *Interaktionen* und *Objekte* Parameter und Attribute besitzen, sind diese typlos. Zumindest interessiert sich HLA nicht für die Typen der Daten, die ausgetauscht werden. Die Interpretation liegt alleine bei den Federates.

Dass Datenklassen von einander erben können, hat auch einen Sinn: Wird die Federation durch neue Federates ergänzt, so können diese mehr als die bisher definierten Attribute oder Parameter benötigen. Erweitert man nun eine alte Klasse durch Vererbung um einige Attribute oder Parameter, so können die alten Federates noch den Teil der Daten der erbenden Klasse mitgeteilt bekommen, der in der vererbenden Klasse schon definiert ist. Bei Definition einer komplett neuen Klasse wäre dies nicht möglich.

Beispiel: Wird ein neuer Federate geschrieben, der die Interaktion IPpacket um einen Parameter *Größe* zur Klasse IPpacketMitGröße erweitert, so können die

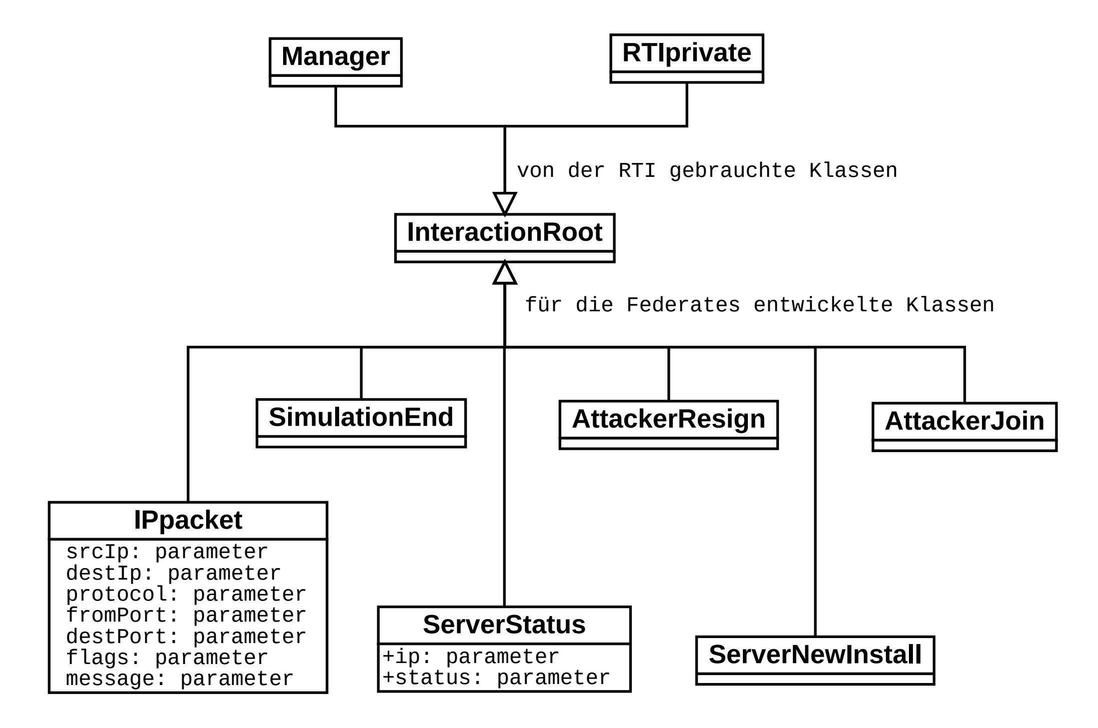


Abbildung 8: Die Hierarchie der Interaktionen

bestehenden Federates immer noch die Interaktion als IPpacket zugestellt bekommen.

Noch ein Hinweis zur Benennung der Klassen: Der offizielle Name einer Interaktion oder eines Objekts wird der Vererbungshierarchie nach gebildet. Dabei wird der Vererbungsbaum von der Rootklasse an abgelaufen, wobei jede Ebene des Baums mit einem "." von der vorigen getrennt wird. Daraus ergibt sich als offizieller Name der Interaktion IPpacket z.B. InteractionRoot.IPpacket, die davon abgeleitete Klasse IPpacketMitGröße würde InteractionRoot.IPpacket.IPpacketMitGröße heißen.

6 Modelle zur Beschreibung von (D)DoS

Ehe man einen Prototyp für eine DoS-Simulation schreiben kann, muss geklärt werden, welches Verhalten die Simulationsteilnehmer in bestimmten Situationen aufweisen sollten. Dieses gewünschte Verhalten wird durch Modelle beschrieben, die jeweils einen bestimmten Aspekt der Simulation beleuchten.

Zur Beschreibung der Modelle wird unter anderem die UML (Unified Modeling Language, [3],[39]) verwendet.

6.1 Verhaltensmodelle für Angriffe

Zuerst wird nun das Verhalten bei den Angriffen beschrieben, die simuliert werden sollen. Sowohl die SYN-Attacke als auch der Smurf-Angriff beruhen auf der Auslastung des Opfers. Untersucht werden muss also das Verhalten des Servers im Falle einer Attacke.

Weiter braucht der Angreifer eine Methode, mit der er die Verteilung seiner Angriffs-Tools erreichen kann. Auch hierfür wird ein Modell benötigt, das den Ablauf von Verteilung und Installation von Tools beschreibt. In diesen Kapitel werden die Vorgänge beschrieben, die bei Anwendung eines Exploits zur Installation der DoS-Tools auf dem Server ablaufen. Die Aktivitäten des Angreifers selbst werden erst mit dem Angreifermodell in Kapitel 6.2 dargestellt.

Hier also das Server-Verhalten bei den unterschiedlichen Angriffen:

Bei einer **SYN-Attacke** werden auf dem Server alle Ressourcen belegt, die zu einem TCP-Verbindungsaufbau von außen benötigt werden. Andere Kommunikation, wie UDP, ICMP oder eine ausgehende TCP-Verbindung, bleiben uneingeschränkt. Das Modell braucht also eine Beschränkung, wieviele eingehende Verbindungen es gleichzeitig verwalten kann. Das sind die Verbindungen, bei denen der Server ein SYN-Paket erhalten hat, und nun auf das ACK wartet. Wird diese Maximalzahl überschritten, so darf das Modell keine weiteren neuen TCP-Verbindungen, das sind solche mit gesetztem SYN-Bit, mehr annehmen.

Ist eine Verbindung nach einer bestimmten Zeit t noch nicht zustande gekommen (das ACK-Paket steht immer noch aus), so wird sie durch Auslösen eines TimeOuts beendet, die von ihr belegten Ressourcen werden freigegeben. Diese Ressourcen werden auch freigegeben, wenn eine Verbindung durch Ankunft des ACK-Pakets zustande kommt. Sobald eine Verbindung ihre Ressourcen wieder freigegeben hat, kann eine neue Verbindung angenommen werden.

Das eben definierte Modell der SYN-Attacke ist in Abb. 9 zu sehen.

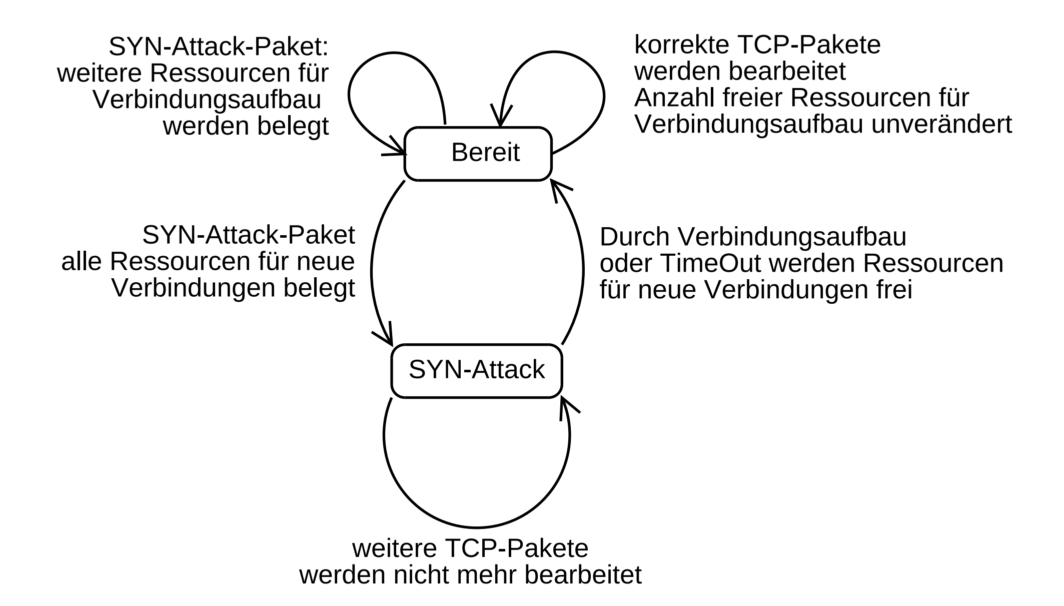


Abbildung 9: Verhalten bei einer SYN-Attacke

Bei **Smurf** ist das Verhalten ähnlich, betrifft aber die gesamte eingehende Kommunikation, unabhängig von dem verwendeten Protokoll oder dem Status der Verbindung. Für die Modellherleitung werden folgende Annahmen getroffen, die ein vereinfachtes Abbild der Realität schaffen: Die Bearbeitung eines eingehenden Pakets beansprucht einen Server $1/n$ Sekunde. Pro Sekunde kann dieser Rechner also n Pakete verarbeiten. Kommen mehr als diese n Pakete an, so können nicht mehr alle bearbeitet werden. Der Rechner ist ausgelastet.

Das Modell soll also festlegen, dass ein Rechner pro Zeiteinheit höchstens n Pakete bearbeiten kann. Alle weiteren Pakete werden nicht berücksichtigt. Ist die Zeiteinheit um, so kann der Rechner wieder n Pakete bearbeiten, bevor er die weiteren wegwirft, usw. Für das Modell der Smurf-Attacke siehe Abb. 10.

In Kapitel 2 wurde erwähnt, dass bei einer DDoS-Attacke der Angreifer mit Hilfe eines **Exploits** Schwachstellen in einem Programm ausnutzen und damit Befehle auf einem fremden Rechner ausführen kann.

Zunächst sendet der Angreifer Daten an ein Programm, um es auszuführen. Wenn dies durch einen Exploit geschieht, so werden diese Daten das Programm zur fehlerhaften Ausführung und so zu der Nebenreaktion bringen, die der Angreifer wünscht. Es wird dann eine Aktion aufgerufen, die der Angreifer mit den von ihm gesendeten Daten bestimmen kann. Allerdings muss das Programm selbst herausfinden können, ob die empfangenen Daten Fehler des Programms ausnutzen oder nicht. Der Ablauf einer Exploitanwendung stellt sich also wie in Abb. 11 dar.

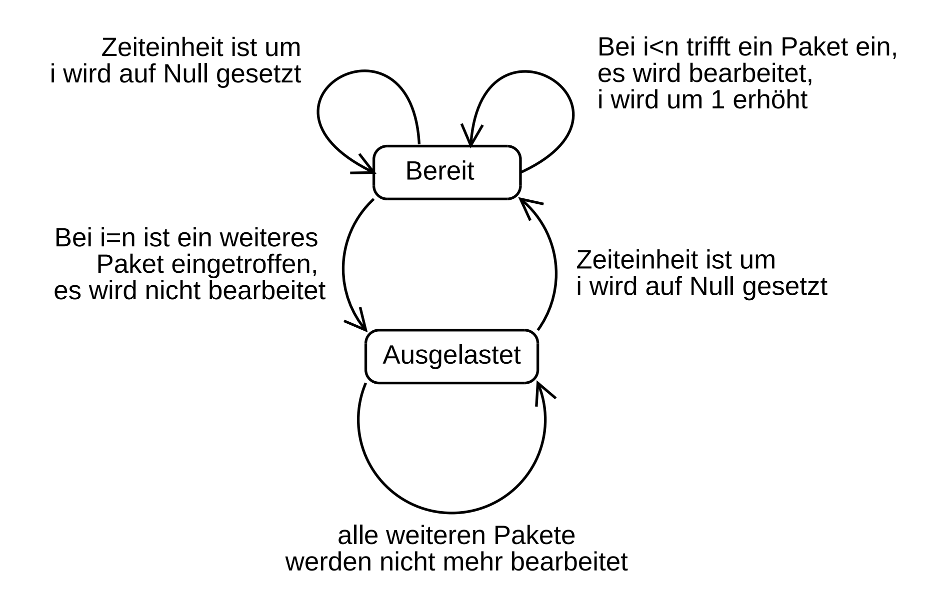


Abbildung 10: Verhalten bei einer Smurf-Attacke

6.2 Angreifermodell

Es gibt kein allgemeingültiges Angreifermodell. Vielmehr muss man für jeden Typ von Angreifer und Angriff ein eigenes Modell entwerfen. Im Folgenden entwickeln wir beispielhaft zwei verschiedene Angreifermodelle. Das erste Modell wird einen Angreifer beschreiben, der eine Smurf-Attacke durchführt, das zweite behandelt einen Angreifer, der zunächst ein Angriffsnetzwerk aufbaut, um dann seinen DDoS-Angriff durchzuführen.

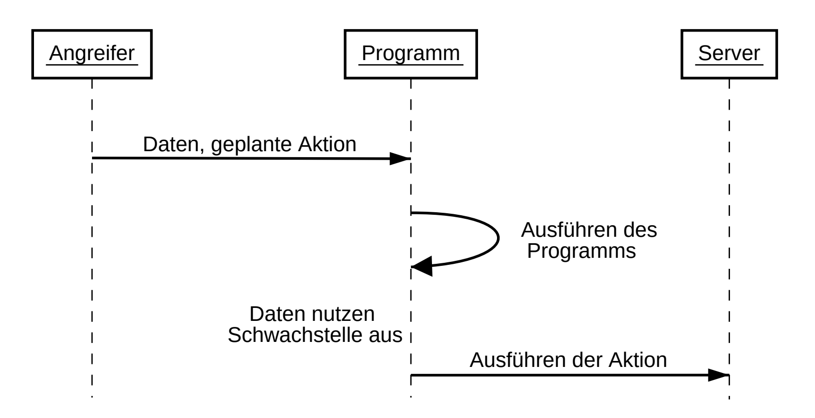


Abbildung 11: Ablauf bei Anwendung eines Exploits

6.2.1 Smurf-Angreifer

Bei einem Smurf-Angriff ist das Verhalten des Angreifers sehr einfach. Er beginnt zu einem bestimmten Zeitpunkt mit dem Angriff auf sein Opfer und beendet ihn eine Zeit danach wieder. Eventuell greift er auf diese Art mehr als ein Opfer an. Wenn der Angreifer keine weiteren Angriffe mehr durchführt, wird seine Betrachtung beendet (Abbildung 12).

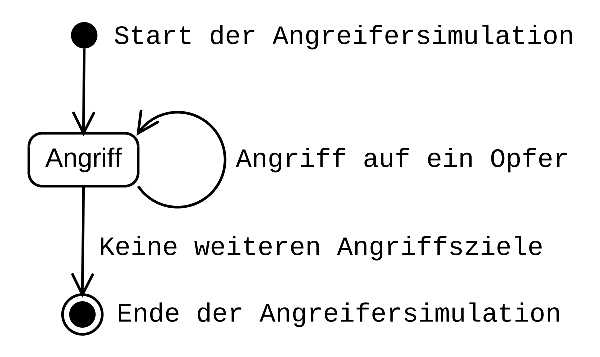


Abbildung 12: Angreifer bei Smurf

Zerlegen wir dieses Modell weiter: Ein Angriff besteht aus der Auswahl eines Opfers und der Entsendung vieler Ping-Pakete auf die Broadcastadresse eines möglichst großen Netzwerks, welches Broadcast-Pings beantwortet. Dieses Netzwerk sollte der Angreifer also vor dem Angriff ausfindig machen. Siehe hierzu Abbildung 13.

6.2.2 DDoS-Angreifer mit Netzwerkaufbau

Um den Angreifer sowohl beim Aufbau seines Netzwerkes als auch beim eigentlichen Angriff zu simulieren, brauchen wir ein Modell, das sämtliche Phasen, die Aktivitäten im Internet beinhalten, berücksichtigt (siehe 2.5.2) und sie sequentiell abarbeitet. Wenn wir das Modell einmal als Zustandsdiagramm (Abbildung 14) darstellen, dann haben die Zustände folgende Bedeutung:

- **Startzustand**
  Unser Angreifer startet in einem Zustand, in dem er über keinerlei Wissen über das Netz verfügt. Er hat nur Zugriff auf das Benutzerkonto, von dem aus er seine weiteren Aktionen durchführen wird.

  Dies ist der Startzustand der Simulation.

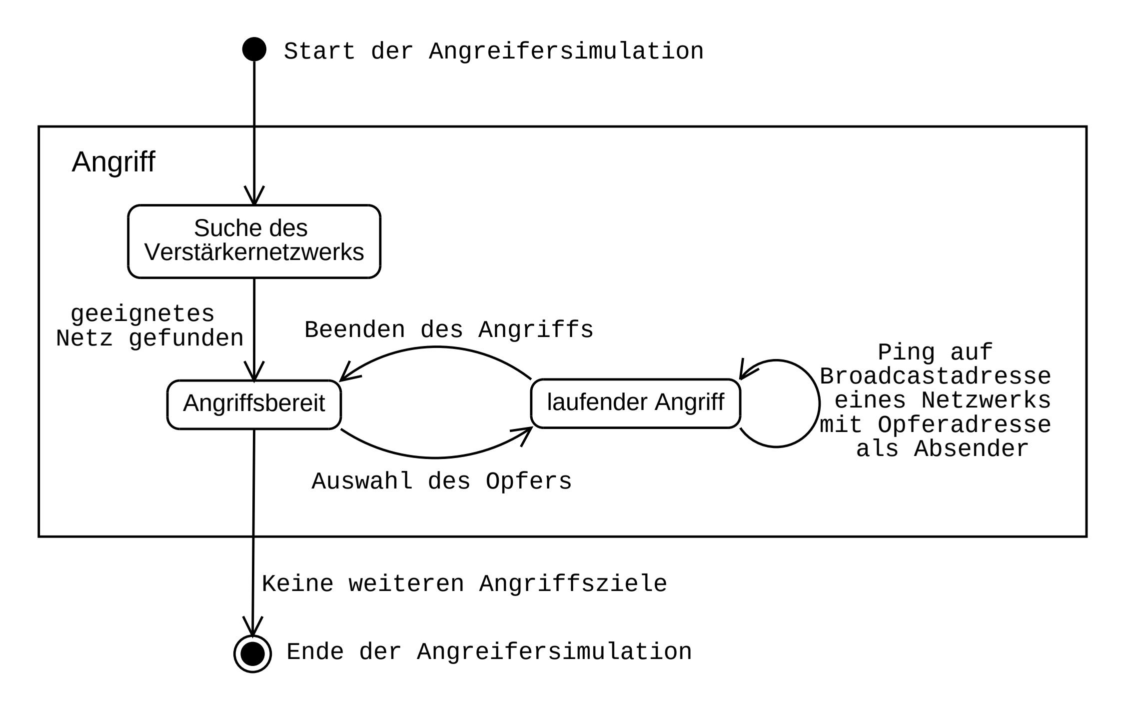


Abbildung 13: Genaueres Angreifermodell

- **Netzscan**
  Der Angreifer scannt das Internet nach potentiellen Helfern für seinen Angriff, d.h. er sucht ganze Netzabschnitte nach Rechnern mit ihm bekannten Sicherheitslücken ab. Wenn er genug Systeme mit Schwachstellen gefunden hat, kann er in den Zustand *Rechnerübernahme* gehen.

- **Rechnerübernahme**
  Nun beginnt der Angreifer damit, sich Root-Zugriff auf den beim Netzscan gefundenen Rechnern mit Sicherheitslücken zu verschaffen. Wenn er genug davon übernommen hat, spätestens aber, wenn er alle davon übernommen hat, geht das Modell in den Zustand *Daemoninstallation* über.

  Während dieses Übergangs zur *Daemoninstallation* legt der Angreifer fest, welche der Systeme Master und welche Daemon werden.

- **Daemoninstallation**
  Die Daemons werden installiert. Wenn sämtliche Daemons fertig eingerichtet sind, geht der Angreifer zur *Masterinstallation* über.

- **Masterinstallation**
  Nun werden die Master installiert. Auch hier findet der nächste Zustandswechsel statt, wenn alle Master installiert sind. Der Angreifer ist dann Angriffsbereit.

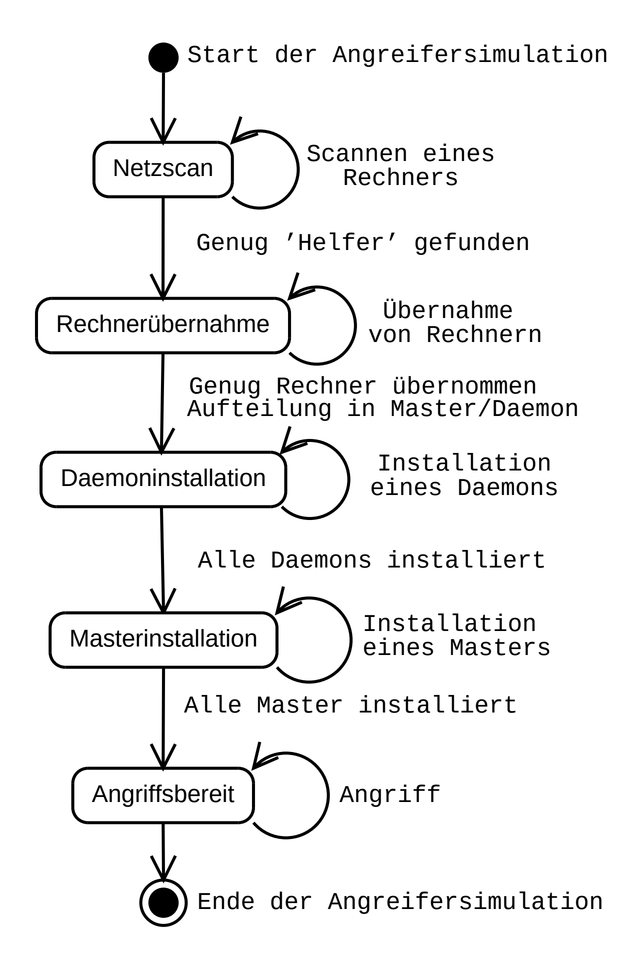


Abbildung 14: Grobes Zustandsdiagramm des Angreifers

- **Angriffsbereit**
  In diesem Zustand ist das Netz fertig aufgebaut. Der Angreifer führt DDoS-Attacken auf die von ihm gewünschten Ziele aus. Wenn das letzte seiner Ziele attackiert worden ist, geht der Angreifer in den Endzustand über. Die einzelnen Angriffsziele müssen nicht alle direkt im Anschluss aneinander angegriffen werden. Vielmehr kann eine zeitlich gesehen große Pause zwischen den einzelnen Attacken sein, und Opfer können auch mehrfach des Ziel einer Attacke werden.

- **Endzustand**
  Der Angreifer hat seinen Angriff beendet. Ein wirklicher Angreifer könnte natürlich weiterhin Angriffe führen. Im Rahmen dieser Arbeit scheint es aber sinnvoll, die Simulation des Angreifers an diesem Punkt zu beenden.

(Je nach Abstraktionstiefe kann gegebenenfalls noch eine Aufteilung der einzelnen Zustände nötig werden. Z.B.: Angriff unterteilt sich in die Angriffsbefehle an die einzelnen Master, Unterschied, wenn Master online/offline, ...)

6.3 Opfermodell

Wir werden zunächst einmal ein einfaches, allgemein anwendbares Modell eines Opfers aufstellen, das nicht speziell auf die Situation des Opfers eingeht. So können die Auswirkungen auf die, die Nutzer des Opfers sind, also in Gegensatz zu Angreifern legale Anfragen stellen, nicht dargestellt werden. Dazu sind die Informationen, die das Opfer über sich und seine Umwelt hat, zu gering. Es sollte dafür ein speziell auf einzelne Server bzw. zumindest eindeutige Servergruppen (Web-Server, Mail-Server, ...) abgestimmtes Opfermodell oder gar ein Opfernetzwerkmodell erstellt werden, dem seine Funktion bewusster ist, als dem allgemeinen Modell.

Das Modell eines Opfers, welches auf jeden Server eines Netzwerks anwendbar ist, nennen wir auch Universal-Opfer.

Das Modell des Universal-Opfers ist noch einfacher als das des Angreifers, da das Opfer nicht aktiv agiert, sondern in unserem Fall nur als passives Element den (Miss-)Erfolg des Angriffs veranschaulichen soll.

Ein universelles Opfer kann folgende Zustände einnehmen:

- **offline**
  Der Rechner ist vom Netzwerk aus nicht erreichbar. In der Regel ist dies während Wartungsarbeiten oder nach einem Absturz der Fall.

- **working**
  Das Opfer befindet sich in funktionsfähigem Zustand, d.h. es ist online und bietet seine Dienste an.

- **out of order**
  Der Opferrechner ist überlastet, weil er gerade angegriffen wird. Aus diesem Zustand kann das Opfer in den *Working*-Zustand übergehen, wenn der Angriff beendet wird und das Opfer danach wieder normalen Betrieb erreichen kann. Sollte das Opfer während des Angriffs abstürzen, so wird es als nächstes in den Zustand *Offline* übergehen, um zu zeigen: Dieser Rechner ist nicht mehr vom Netz aus zu erreichen. Sollte das Opfer nach Beendigung des Angriffs nicht in den normalen Betrieb zurückkehren können, weil Teile des Systems durch den Angriff ausgefallen sind, so wird es solange in *out of order* bleiben, bis ein Reboot erfolgt. Durch diesen wird wieder Betriebsbereitschaft hergestellt.

Das Diagramm zu diesem Universal-Opfer-Modell ist in Abbildung 15 dargestellt.

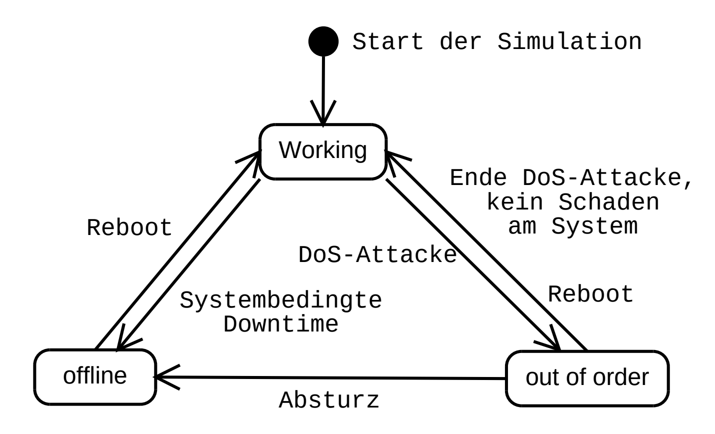


Abbildung 15: Zustandsdiagramm des Opfers

Solange man die Angriffsauswirkungen nicht genauer untersuchen will, reicht dieses Modell vollkommen aus, um bei einem DoS-Angriff die Auswirkungen zu zeigen. Wenn man aber etwa zeigen will, wie sich der Angriff aus Sicht der User auf die Erreichbarkeit des Systems auswirkt, dann ist dieser Modellansatz nicht ausreichend.

6.4 Modell der Netzwerkrechner

Dieses Modell beschreibt die Rechner des Internets, von denen der Angreifer einige zu seinen Helfern machen wird. Diese Rechner können in verschiedene Klassen eingeteilt werden:

- **Wichtige Server**

  Diese Server besitzen eine hohe Bandbreite und eine feste IP-Adresse und sind für den Betrieb zumindest in ihrem Teilnetz so wichtig, dass man sie selbst bei einem Verdacht auf DoS-Aktivitäten nicht so schnell aus dem Netz nimmt. Bei einem Ausfall dieser Server wird die gesamte Internet-Aktivität oder zumindest die Aktivität einzelner Netzbereiche stark eingeschränkt.

  Zu dieser Servergruppe gehören beispielsweise Primary-Name-Server.

- **Server**

  Dieses sind Rechner, die dauerhaft im Internet erreichbar sind, aber von nicht so entscheidender Bedeutung. Bei Verdacht auf DoS-Aktivitäten könnten diese Server leicht aus dem Netz genommen werden, ohne andere Server oder

Netzaktivitäten stark zu beeinträchtigen. Die einzigen gestörten Verbindungen sind die, die direkt mit diesem Server bestehen sollen.

- **Privatrechner**

  Diese Rechner besitzen keine feste IP-Adresse. Sie sind meistens nicht gut gegen Angriffe geschützt, sind aber im Rahmen einer klassischen DoS-Attacke schlecht brauchbar, weil sie eben nicht ständig verfügbar sind und man sie auch schlecht wiederfindet, wenn sie eine neue IP zugewiesen bekommen haben. Als Daemon wären sie dennoch brauchbar, aber dazu sind die meisten dieser Rechner zu kurzzeitig im Internet.

  Für diese Arbeit ist diese Rechnergruppe aus obengenannten Gründen uninteressant, ein Angreifer wird bei seiner Suche solche Rechner möglichst gleich auslassen. In der Simulation werden wir Privatechner deshalb nicht berücksichtigen.

Es gibt keine klaren Grenzen zwischen den eben angegebenen Klassen, die Übergänge können sehr fließend sein. Ebenso ist es auch gut möglich, eine andere Einteilung in Klassen zu finden. Wir haben diese gewählt, da sie für die Sicht des Angreifers zur Auswahl der potentiellen Helfer geeignet erscheint.

In unserem Modell unterscheiden sich die Rechner verschiedener Rechnerklassen durch die Verweildauer in bestimmten Zuständen und durch die Häufigkeit von Zustandsübergängen. Der Zustandsautomat von Rechnern verschiedener Klassen ist identisch.

Ein durchschnittlicher Server im Internet verhält sich wie folgt:

Der Server ist die meiste Zeit online verfügbar und stellt seine Dienste zur Verfügung. Ab und zu muss er aus systemtechnischen Gründen vom Netz genommen werden. Ob dies durch einen Absturz, eine Neuinstallation, Hardwarearbeiten oder sonstiges bedingt wird, spielt aus Netzsicht keine Rolle. Der Effekt bleibt gleich, der Rechner wird für andere Netzteilnehmer unerreichbar. Was der Grund für die Downtime[11] war spielt nur für den Zustand eine Rolle, in den der Server eintritt, wenn er wieder verfügbar wird. Nach einer Neuinstallation z.B. ist das System in sauberem Zustand, d.h. der Server ist online, und eventuell vorher installierte Schadprogramme (Viren, Trojanische Pferde, Hintertüren, aber auch Master- oder Daemonprogramme) sind nun nicht mehr auf dem System zu finden.

Aber neben dem Normalbetrieb gibt es in Hinsicht auf DDoS-Simulationen noch Zustände, in denen der Server weiter seiner Arbeit nachgeht, aber eben nicht mehr „sauber” ist. Der Server kann von einem Hacker übernommen worden sein.

[11] Als Downtime wird die Zeitspanne bezeichnet, die der Server vom Netz getrennt ist

Wenn dieser die Serverkonfiguration manipuliert hat, z.B. eine Hintertüre eingerichtet hat, mit der er sich jederzeit mit Administratorrechten auf dem System einloggen kann, so ist der Rechner nicht mehr „sauber”. Dieser Zustand wird im Modell „übernommen” genannt. Aber auch durch die Installation von Programmen (Viren, Würmer, Trojaner, ...) kann der Server in den Zustand „übernommen” gelangen.

Ein vom Angreifer übernommener Rechner unterscheidet sich dann noch darin, ob die für den Angriff nötigen Programme installiert sind.

Fassen wir also die Serverzustände zusammen:

- **offline**
  Das System ist nicht erreichbar. Eventuell wird das System neuinstalliert, dann wird es bei Verlassen des *Offline*-Zustands in den Zustand *saubere Installation* übergehen. Ansonsten wird das System nach der Downtime wieder in den Zustand übergehen, in dem es war, bevor es Offline gegangen ist.

- **saubere Installation**
  Das System ist in einem Zustand, in dem der Angreifer keinen Root-Zugriff darauf besitzt.

- **übernommen**
  Der Angreifer besitzt Root-Zugriff auf den Server und kann daher alles mit dem System machen, was er will. Dazu zählt u.a. Programminstallation, Belauschen des Netzwerkverkehrs oder das Ändern der Systemkonfiguration.

- **installierter Master**
  Auf dem Server sind die Programme zur Ausführung der Mastertätigkeit funktionsfähig. Es läuft ein Programm, das auf Befehle des Angreifers wartet, um diese auszuführen.

  Aus diesem Zustand kann den Server nur eine Downtime mit Neuinstallation herausbringen.

- **installierter Daemon**
  Auf dem Server sind die Daemon-Programme installiert und einsatzbereit. Der Daemon wartet auf Befehle des Masters. Auch aus diesem Zustand kann der Server nur durch eine Neuinstallation herausgebracht werden.

Die Zustände samt Übergängen sind in einem Diagramm in Abbildung 16 dargestellt.

Hinweis: Diese Zustände beziehen sich immer auf die Sicht eines speziellen Angreifers, denn im Normalfall sind die installierten Tools für die Angriffe z.B. Passwort-geschützt, damit kein anderer Angreifer sie benutzen kann. Durch häufigere Nutzung würde das Risiko des Entdecktwerdens gesteigert und damit der Ausfall dieses „Helfers” wahrscheinlicher.

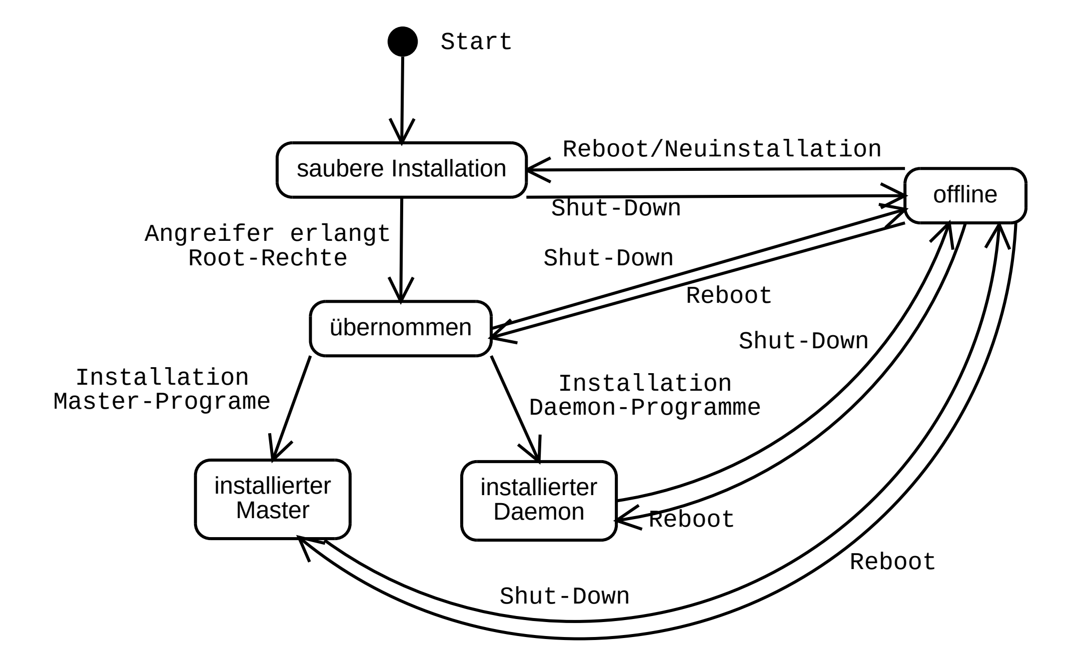


Abbildung 16: Zustandsdiagramm eines Netzrechners

6.5 Netzwerkmodell

Will man das Internet simulieren, so reicht es nicht, die einzelnen Rechner, die daran beteiligt sind, zu simulieren. Man muss sich ebenso überlegen, wie diese Einzelteile zusammenarbeiten, wie sie kooperieren und kommunizieren.

Dazu gehört, die Netzwerkverzögerungen einzuarbeiten. Ein Rechner darf beispielsweise nicht zu der Zeit, zu der er seine Anfrage an einen Server stellt, auch schon die Antwort des Servers erhalten. Wenn wir unser Netzwerk in festen Schritten takten würden, so darf nur ein Schritt einer Kommunikation in einen Zeitabschnitt fallen.

Beispiel: Im ersten Schritt sendet der Client seine Anfrage an den Server. Im zweiten bearbeitet der Server die Anfrage und generiert eine Antwort. Im dritten Schritt kann der Client die Antwort auswerten.

Veranschaulicht wird dieses Verhalten in Abb. 17, die horizontalen gestrichelten Linien symbolisieren dabei den Schritt von einem Zeittakt zum nächsten.

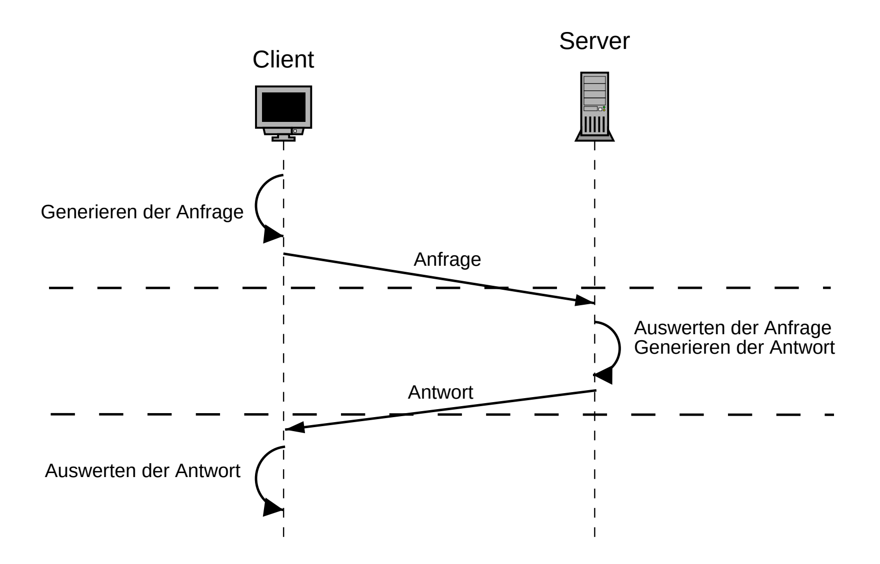


Abbildung 17: Ablauf einer Kommunikation

7 Konzept für die Implementierung

Um ein gutes Konzept zu entwickeln, sollte erst ein Anforderungskatalog erstellt werden, den das zu entwickelnde System später weitgehend erfüllen sollte. Hierbei gibt es „Muss"-Kriterien, die unbedingt enthalten sein müssen, und „Soll"-Kriterien, die wünschenswert sind, aber nicht notwendig.

Die Wesentlichen Anforderungen, die an den Prototyp einer DoS-Simulation gestellt werden, sind folgende:

- **Simulation einer DoS-Attacke muss möglich sein**
  Ein unbedingt notwendiger Punkt, denn man kann von keiner DoS-Simulation reden, wenn man keinen Angriff damit simulieren kann.

- **Erweiterbarkeit**
  Ein sinnvolles Kriterium, das man bei der Entwicklung eines Prototyps beachten sollte. Denn wenn man den Prototyp später noch erweitern kann, damit auch andere als die mit ihm zusammen implementierten Attacken dargestellt werden können, ist das für auf dem Prototyp aufbauende Arbeiten hilfreich.

- **einfache Benutzbarkeit**
  Um einfaches Erweitern und Anwenden des Prototyps zu ermöglichen, sollte kein Spezialwissen erforderlich sein. So sollte versucht werden, einfache Schnittstellen bereitzustellen, mit denen z.B. HLA-Aufrufe versteckt werden können.

7.1 Netzwerktopologie

Wenn ein Angriff simuliert werden soll, gibt es einen Angreifer, der ein Opfer attackiert. Im Falle einer verteilten Attacke werden noch Helfer für den Angriff hinzugezogen, die aus Sicht des Angreifers irgendwo im Internet zu finden sind. Hieraus ergibt sich eine mögliche Gliederung nach Angreifer, Opfer und Internet. Es ist aber nicht sinnvoll, ein so komplexes Gebilde wie das Internet in einem Block abzuhandeln. Es wäre sicherlich möglich, aber dadurch gehen viele Aspekte des Internets unnötigerweise verloren. Viele Funktionen, die Routerbetreiber bei der Abwehr von DoS-Angriffen ausüben könnten, wären nicht oder nur sehr vermindert darstellbar. Gerade einfache, wichtige Techniken, wie das Filtern von offensichtlich gespooften Paketen, die von den Helfern im Internet abgeschickt würden, ließen sich mit diesem zu einfachen Konzept schlecht darstellen.

Stellen wir uns deshalb das Internet als Netz von Netzen vor. Damit kann man jedem dieser Teilnetze einen Router zuordnen, der den Paketverkehr aus dem oder

in das Netz kontrollieren kann. Bei Verwendung von nur einem solchen Teilnetz ist das Konzept so einfach wie das vorherige, weil dieses eine Teilnetz das gesamte Internet abdecken müsste, aber schon bei Verwendung von zwei Netzen hat man deutlich mehr Möglichkeiten.

Weiterhin braucht man bei diesem Ansatz keine Sonderbehandlung für die Kommunikation mit Angreifern oder Opfern. Diese kann man vorerst wie eigenständige Teilnetze behandeln. Man kann aber auch später durch Entwicklung eines neuen Typs von Teilnetz gezielter auf die Situation spezieller Gruppen wie z.B. ISPs (Internet Service Provider) eingehen, die verhindern wollen, dass ihre Nutzer DoS-Angriffe auf Dritte führen.

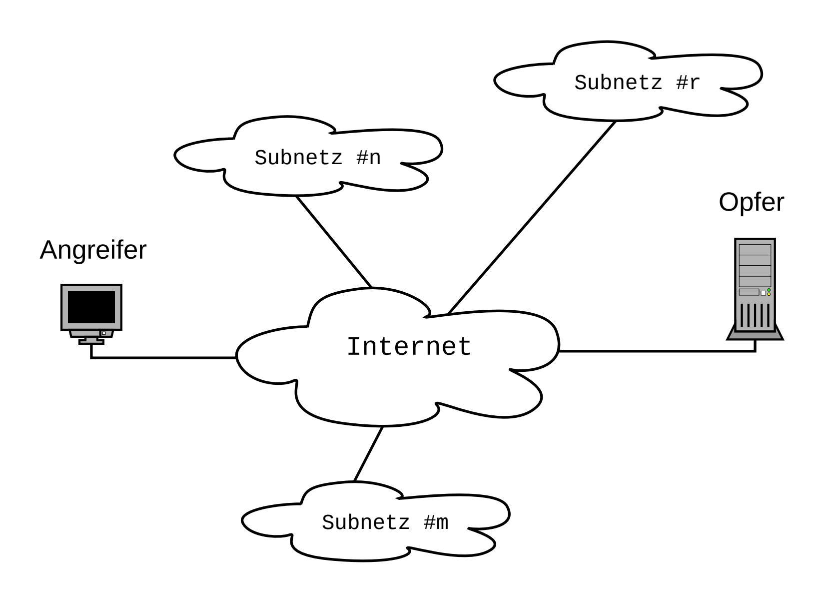


Abbildung 18: Konzept im Kontext der realen Welt

Der Prototyp wird sich aber zunächst neben Angreifer und Opfer auf einen Typ von Teilnetz beschränken. Illustriert wird dieses Konzept in Abb. 18. Das eigentliche „Internet” dient hier nur als Vermittler, der die Kommunikation zwischen den einzelnen Teilnetzen, Angreifer und Opfer ermöglicht. Im Gegensatz zum realen Internet kann hier also jedes Teilnetz mit jedem Teilnetz direkt kommunizieren. Bei dem Prototyp übernimmt diese Aufgabe die RTI (Runtime Infrastructure), die von HLA bereitgestellt wird. Die Teilnetze sind dann ebenso wie Angreifer und Opfer eigenständige Federates (Abb 19).

7.2 Teilnetze

Auch bei den Teilnetzen gehen wir davon aus, dass jeder Rechner direkt mit jedem anderen Rechner des Teilnetzes verbunden ist. Jede Kommunikation, die mit Rech-

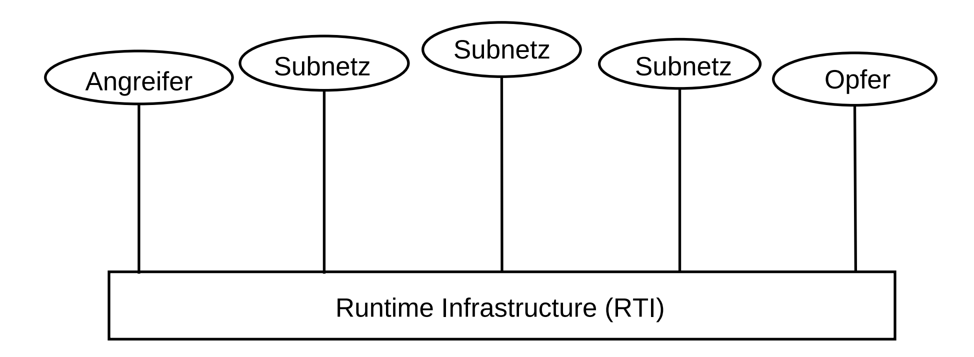


Abbildung 19: Konzept im HLA-Kontext

nern außerhalb des eigenen Teilnetzes durchgeführt wird, läuft über einen Router. Dieser kann gleichzeitig als Firewall dienen. Während im realen Internet die Funktion von Router und Firewall manchmal aus Effizienzgründen nicht auf demselben Rechner läuft, brauchen wir uns um diesen Aspekt keine Gedanken zu machen, da der Verkehr des simulierten Netzes nicht so hoch sein wird, wie ihn stark frequentierte Router im realen Internet bewältigen müssen. Außerdem würde es für das Teilnetz, welches als Ganzes durch einen Federate dargestellt wird, keine Effizienzsteigerung bringen, diese Funktionen zu trennen.

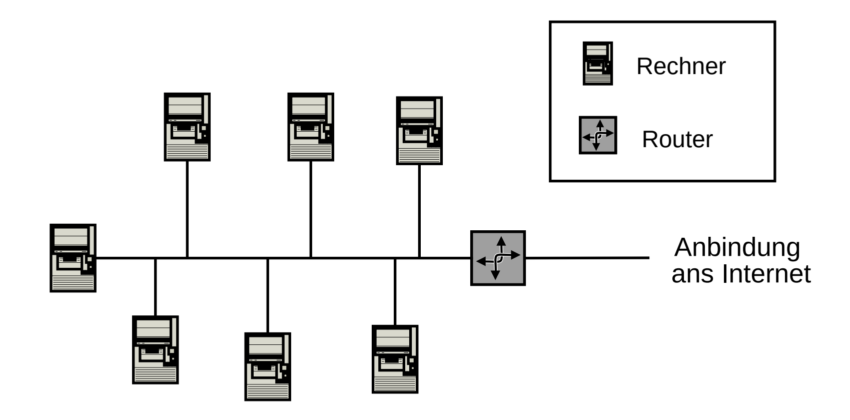


Abbildung 20: Konzept eines Subnetzes

Bei der Konzeption der Subnetze hat sich die Frage gestellt, ob es sinnvoll ist, das Netz nur aus den Servern und dem Router bestehen zu lassen, oder ob zusätzlich eine die Netzstruktur (Bussystem, Kabel und Switch oder ähnliches) repräsentierende Instanz eingeführt werden soll. Die Entscheidung fiel zu Gunsten einer Repräsentierung der Netzstruktur, da so viele zusätzliche Möglichkeiten simulierbar werden, die sonst wegfielen. Durch eine solche Instanz kann z.B. für Bus-Netze (BNC) eine Kollisionssimulation eingeführt werden, man kann Verzögerungen des

Netzverkehrs nachbilden, etc. Auch wenn diese Möglichkeiten in dem Prototyp nicht ausgenutzt werden, so stellen sie doch Anreize für zukünftige Erweiterungen dar.

Ein weiterer Grund für diese Entscheidung war, dass auf diese Art die Server sich untereinander nicht kennen und ansprechen können müssen. Die Server können also unabhängiger voneinander agieren. Für sie ist es so kein Unterschied, ob sie mit einem Rechner aus ihrem Netz kommunizieren oder Pakete über das Internet versenden. Sie müssen so auch ihre Entscheidungen zum Paketverkehr im Intranet nicht danach treffen, ob ein anderer Server erreichbar ist oder nicht.

7.3 Kommunikation im Netzwerk

Im Rahmen der DoS-Simulation durch den Prototyp kann auf eine Darstellung von Netzwerkverkehr, der nicht an der Darstellung der DoS-Attacke beteiligt ist, verzichtet werden. Dieser Verkehr macht nur dann Sinn, wenn man die Auswirkungen des Angriffs auf die regulären Nutzer des Opfers simulieren will. Aber auch dann sollte man darauf achten, eine Art der Netzlasterzeugung zu benutzen, die auch in Hinsicht auf reale Netzlast sinnvoll ist.

Da in unserer Simulation dieser Netzverkehr (noch?) nicht wirklich zu Auswirkungen auf das Netzverhalten führt und keine Auswirkungen auf den Angriffsverlauf ausübt, nimmt uns seine Simulation höchstens Ressourcen zur Berechnung der übrigen Simulation weg. Weiter ist es schwer, mit einem so kleinen Ausschnitt des Internets, welches wir in den Beispielsimulationen verwendet haben, realistischen Verkehr zu erzeugen. Und um einen Federate zu schreiben, der sich um Trafficerzeugung aus nicht eigenständig simulierten Netzen kümmert, ist in dieser Arbeit keine Zeit verblieben. Dies ist auch eine schwierige Aufgabe, wie schon in Kapitel 4 oder in [18] erwähnt wird.

Für den von uns benötigten Traffic scheint es sinnvoll zu sein, ihn an die im richtigen Internet verwendeten Protokolle anzulehnen, d.h. Unterscheidungen zwischen TCP, UDP und ICMP zu machen. Gerade wenn man Firewalls simulieren möchte, sollte der Unterschied der Protokolle zum Tragen kommen können. Ebenso ist es schwierig, eine TCP-SYN-Attacke ohne eine Repräsentierung des TCP-Protokolls zu ermöglichen.

Die nächste Frage, die sich stellt, ist, wie diese Protokolle dargestellt werden sollen. Der Ansatz, die Kommunikation wie im realen Internet in Pakete einer maximalen Größe zu unterteilen, wird schnell verworfen, da er große Probleme bei der zeitlichen Verwaltung der Simulation aufwirft und zu aufwendig erscheint, um ihn hier umzusetzen. Also wird pro Nachrichtenblock ein Paket verwendet.

Damit ist die Konzeption für ICMP und UDP beendet. Für die TCP-Verbindung wurde zunächst der Ansatz verfolgt, wie in der realen Welt die Pakete immer mit Rückpaketen zu bestätigen. Der Verbindungsaufbau (Handshake) könnte so auch mit drei Paketen nachgebildet werden. Die TCP-Flags der Pakete der simulierten Verbindung wären so wie beim realen TCP zu setzen.

Dieser Ansatz wurde wieder verworfen, da er sich als nicht praktikabel herausstellte. Das Timing des Paketverkehrs war nicht mit dem der restlichen Simulation vereinbar, eine TCP-Verbindung wäre so im Verhältnis viel zu langsam geworden. Aber auch ein zweiter Grund sprach gegen diesen Ansatz: Er lief auf einen großen Implementierungsaufwand hinaus, weil ein simulierter Server so wie ein realer sich die Zustände, in denen sich eine TCP-Verbindung befindet (im Aufbau, zustande gekommen, etc.) merken müsste. Ebenfalls wäre zusätzlicher Aufwand bei der Bearbeitung der empfangenen Pakete nötig geworden.

So wurde ein einfacheres Konzept entwickelt, wie man eine TCP-Verbindung darstellen kann. Dabei verlassen wir uns darauf, dass die RTI alle Interaktionen wirklich weitergibt. Somit können wir auf die Bestätigungs-Pakete verzichten. Der Status der Verbindung wird durch die TCP-Flags codiert, die in den Paketen gesetzt werden. Damit reicht auch hier ein Paket für jede Verbindungsrichtung, genau wie bei UDP. Genaueres zu diesem Konzept ist der Beschreibung der Implementierung (8.2.6) zu entnehmen.

7.4 Konzept eines Servers

Die Server sind sehr wichtig für unsere Simulation, denn sie müssen uns die Funktionalität bieten, die für eine DoS-Attacke gebraucht wird. Sie müssen auf SYN- und Smurf-Attacken reagieren (also bei Auslastung weitere Verbindungen ablehnen), sie müssen als Server Programme anbieten, die andere von außen erreichen können, um sie zu benutzen. Und weiter soll es möglich sein, dass man auf unterschiedlichen Servern auch unterschiedliche Programme laufen lassen kann.

Die Programme müssen sogar während der Simulation installiert und deinstalliert werden können, sonst könnte ein DDoS-Anwender z.B. nach seinem Einbruch in das System nicht die Programme installieren, die er für seine Attacke braucht.

Die Programme sollten also kein fester Bestandteil des Servers sein, sondern eine modulare Ergänzung zu einem Server darstellen. Wir werden deshalb die Programme als Klassen definieren, und jeder Server wird eine Liste der auf ihm installierten Programme erhalten. Diese Liste kann dann während des Simulationslaufs verändert werden, um Installationen und Deinstallationen zu ermöglichen.

Wenn die Programme nun als eigenständige Objekte von einem Server verwaltet werden, so muss ein Server seinen Programmen auch die Möglichkeit bieten, über ihn die Netzwerkkommunikation zu betreiben. Er muss also Funktionen bereitstellen, mit denen die Programme IP-Pakete verschicken können.

Weiter wird eine Möglichkeit gebraucht, ankommende Pakete an die richtigen Programme weiterzuleiten. Hierzu wird ein Portsystem wie auf realen Servern nachgebildet, bei dem sich ein Programm an einen freien Port bindet, um die Pakete zu erhalten, die an diesen Port gesendet werden.

Aber auch Programme, die Pakete abschicken wollen, müssen sich an einen Port binden, damit sie ihren Absende-Port angeben können, und eventuelle Antworten auf ihr Paket zu erhalten.

Bei der Ausführung von Programmen gibt es in der realen Welt mehrere Möglichkeiten:

1. Ein Programm ist ein Dienst, z.B. ein Webserver, der, sobald er gestartet wurde, ständig an einem Port lauscht. Kommt ein Paket an seinem Port an, so wird das Programm mit den empfangenen Daten benutzt. Diese Programme müssen explizit beendet werden, wenn man nicht mehr möchte, dass sie laufen.

2. Ein Programm wird, nachdem es gestartet wurde, ständig ausgeführt. Für unser Szenario bedeutet dies, dass das Programm in jedem Zeittakt vom Server einen Befehl erhalten muss, damit es seine anfallende Arbeit erledigt. Auch hier ist ein expliziter Befehl zum Abbrechen der Ausführung notwendig. Zu dieser Sorte von Programm gehören z.B. Intrusion-Detection-Tools.

3. Andere Programme werden nur kurzzeitig ausgeführt. Man ruft sie auf, sie brauchen eine gewisse Zeit, bis sie ihre Arbeit erledigt haben, dann beenden sie sich automatisch. Dabei können solche Programme regelmäßig automatisch zur Ausführung gebracht werden (z.B. per „cron” unter Unix), oder sie werden von einem User gestartet und damit unregelmäßig benutzt.

Diese Möglichkeiten sollte unser simulierter Server auch bieten. Während die ersten beiden Möglichkeiten in das Konzept des Servers einfließen, ergibt sich bei der dritten ein Problem: Wie sollen wir diese „Interaktivität” nachbilden? Die regelmäßige Ausführung könnte man durch ein Programm simulieren, welches den erforderlichen Programmaufruf zu einer definierten Zeit (hier müsste dann die Simulations-Zeit als Referenz verwendet werden) ausführt.

Aber wie können wir das „unregelmäßige Ausführen”, wie es ein User machen würde, nachbilden? Hierzu wurde ein Konzept angedacht, welches einen User als

eigenständigen Federate simuliert. Er würde sich an einem Rechner einloggen und dort seine Arbeit erledigen. Man würde also in der großen Netzwerksimulation kleine Simulationen für die Usertätigkeit an Rechnern einbringen. Ebenso könnte man solche User sowohl lokal an einem Rechner simulieren als auch ihnen die Möglichkeit geben, remote an einem anderen Rechner zu arbeiten.

Zu diesem Zweck hat der Server einige user-spezifische Funktionen erhalten, wie einloggen, ausloggen, Passwort ändern, u.a. Ebenso wurde vorgesehen, dass Befehle mit einer UserID des Nutzers aufgerufen werden, wobei die UserID 0 für Systemaufrufe, bzw. Administrator-Tätigkeit reserviert war.

Dieser Ansatz wurde nach einiger Zeit verworfen, da er in der verfügbaren Zeit nicht realisierbar war. Selbst wenn man den User nicht als Simulation, sondern stark vereinfacht als statisch in die Anwendung eingebundene Befehlsfolge programmiert, ergeben sich daraus eine Vielzahl von Forderungen an den Server, der dadurch einfach zu komplex wird. Jedes Programm hätte erlaubte Ausführungsrechte beinhalten müssen, die sich dann auch noch danach richten, wer das Programm installiert hat. Daher hätten diese Rechte während der Installation auf dem Server beeinflusst werden müssen. Ein Einbrecher in das System hätte dann auch eine ID erhalten müssen, unter der er während seines Einbruchs agieren kann. Und es hätten alle Aktionen wie Installation, Ausführung von Programmen immer wieder darauf überprüft werden müssen, ob sie überhaupt stattfinden dürfen, usw.

Daher wurde das Userkonzept verworfen und die Ausführung von Befehlen auch ohne eine Angabe der UserID ermöglicht. Die Server spiegeln so etwa ein Windows-98 System wider, bei dem jeder User alles darf. Allerdings kommen hier noch Serverfähigkeiten hinzu.

7.5 Weitere Überlegungen

Nachdem wir nun die Komponenten, aus denen die Netzwerke bestehen sollen, beschrieben haben, kommen wir noch einmal auf die Gesamtsimulation zurück.

Durch das Verwenden von Teilnetzen können wir auch Angreifer und Opfer als solche Teilnetze darstellen. Die Basis, von der der Angreifer seinen Angriff ausführt, ist ein Rechner wie jeder andere in den verschiedenen Teilnetzen. Er hat nur andere Programme installiert, aber er ist genauso angreifbar wie jeder andere Rechner auch.

Das Opfer brauchen wir gar nicht als spezielles Opfer zu entwerfen, wenn wir mit jedem Netzrechner gleich die Funktionen bereitstellen, die ein Angreifer für seinen Angriff benötigt. Damit können in der Simulation beliebige Rechner als Opfer ausgewählt werden.

Um die Aufrufdetails der RTI zu „verstecken”, wird eine Klasse entworfen, die den Servern, Routern oder dem Netz diese Funktionalität bietet. Wenn allerdings mehr als die bereitgestellten Funktionen dieser Klasse gebraucht werden, muss sich der Entwickler wohl oder übel doch mit der Aufrufsyntax der RTI zum Versenden von Interaktionen oder Objekten auseinander setzen, das ist dann nicht zu vermeiden.

Das Zeitverhalten der Simulation wird „konservativ” sein, d.h. jeder Federate darf erst zu einer neuen Zeit voranschreiten, wenn (durch die RTI) sichergestellt ist, dass er keine Events mehr erhalten kann, die vor dieser dann aktuellen Zeit stattfanden.

Auf das Benutzen von Synchronisationspunkten (siehe 5.2) wird verzichtet. Das Internet ist ein derart dynamisches und vitales Gebilde, dass es nicht sinnvoll erscheint, erst alle Federates an einem Punkt synchronisieren zu lassen, bevor die Simulation beginnt. Vielmehr soll es möglich sein, dass ein Netzwerk längerfristig simuliert wird und während dieser Simulation neue Teilnetze hinzugefügt oder alte entfernt werden können (z.B. kommende und gehende Angreifer).

8 Implementierung eines Prototyps

Nachdem in Kapitel 7 eine Gliederung der Gesamtsimulation konzipiert wurde und einige Anforderungen an die entstandenen Einzelkomponenten entwickelt wurden, werden wir uns nun mit der Implementierung der einzelnen Glieder der Simulation befassen und auf weitere Notwendigkeiten und besondere Aspekte ihrer Implementierung eingehen.

Für die Implementierung wurde C++ ([41]) als Programmiersprache gewählt.

8.1 Beenden der Simulation

Im Rahmen unseres Prototyps kann man sich mehrere Möglichkeiten vorstellen, wie man einen Simulationslauf beenden kann. Für jedes Simulations-Szenario sollte getrennt entschieden werden, welche Abbruchmethode gewählt wird. Hier nun einige Möglichkeiten:

- Wenn nur ein Angreifer simuliert werden soll, so kann dieser nach Beendigung seines Angriffs das Signal geben, die Simulation abzubrechen. Hierzu kann er die Interaktion SimulationEnd verwenden. Diese Interaktion sorgt dafür, dass alle Intranetze ihre Ausführung beenden. Wer trotz mehrerer Angreifer diese Interaktion benutzt, sollte sich im Klaren sein, dass die anderen Angreifer bei Empfang von SimulationEnd nicht in jedem Fall auch ihre Ausführung beenden, sondern nur dann, wenn das für sie so implementiert wurde.

  Man kann auf diese Weise auch mehrere Angreifer simulieren, aber man sollte dabei beachten, dass nur der, der als letzter seinen Angriff beendet, diese Interaktion senden sollte, da sonst die anderen Angreifer kein Internet mehr zum Agieren haben, weil alle Subnetze sich aus der Simulation zurückgezogen haben. Kann man nicht genau vorhersagen, welcher Angreifer zuletzt fertig wird, da die Ausführungsdauer des Angriffs vielleicht von dem Verhalten der Simulation abhängt, sollte man eine andere Möglichkeit des Simulationsabbruchs vorziehen.

- Für den Fall, dass mehrere Angreifer aktiv werden sollen, kann eine andere Art von Interaktion verwendet werden. Ein Angreifer, der an der Simulation teilnimmt, kann dies mit der Interaktion AttackerJoin den Intranetzen bekannt geben. Wenn er seinen Angriff beendet hat und sich abmeldet, so sendet er die Interaktion AttackerResign. Damit wissen die Netze, dass sie auf diesen Angreifer nicht mehr warten müssen. Wenn sich also alle Angreifer, die sich irgendwann angemeldet haben, wieder abgemeldet haben, wird die Simulation beendet.

- Die Simulation kann unabhängig vom Status der Angreifer nach einer gewissen Zeit beendet werden.

  Diese Art des Abbruchs ist im Prototyp nicht implementiert, könnte aber auf einfache Weise ergänzt werden. Eine Implementierung hat nicht stattgefunden, weil sich diese Simulation an den Angreifern orientiert. Für eine Simulation, bei der das Verhalten des Internets im Vordergrund steht, nicht der einzelne Angriff, wäre dieser Abbruch aber sinnvoll.

- Es kann eine Mischform zwischen den bisher genannten Varianten eingeführt werden, die den Abbruch nach spätestens einer vorgegebenen Zeit einleitet, ihn aber auch früher schon durch die Interaktion eines Angreifers durchführen kann.

- Ein eigenständiger Management-Federate könnte implementiert werden, der auch die Abbruchkriterien verwaltet. Dieser könnte sowohl die hier schon beschriebenen Methoden zum Bestimmen des Simulationsendes verwenden, als auch andere Kriterien, beispielsweise eine interaktive Möglichkeit, per Mausklick die Simulation zu beenden. Von den anderen Federates sollten dann AttackerJoin und AttackerResign nicht mehr ausgewertet werden, aber alle sollten dann auf den Empfang von SimulationEnd mit ihrer Beendigung reagieren.

8.2 Netzwerk

Ein Netzwerk (auch: Intranet) besteht aus mindestens einem Server, einem Netz und einem Router, der gleichzeitig die Firewallfunktion übernimmt. Darunter liegt das Anwendungsprogramm, das die Zeit verwalten muss und dafür zuständig ist, das Netzwerk zu initialisieren. Es muss eine NetzID beschaffen, das Netz damit erzeugen, dann den Router erschaffen und ihm das Netz bekannt machen und die Server samt ihrer Konfigurationen erstellen. Dieses Anwendungsprogramm kann sowohl interaktiv gestaltet als auch fest vorgegeben werden. Interaktiv heißt hierbei: man kann die Zahl der Server verändern, ihre Konfiguration beeinflussen, gezielt Server herunterfahren bzw. wieder booten. Und man kann die Regeln der Firewall während der Simulation verändern. Zu überlegen wäre, ob man eine „interaktive Firewall" ermöglicht, bei der der User des interaktiven Programms für jedes Paket von außen oder nach außen entscheiden darf, ob es durchgelassen wird oder nicht. Hiermit könnte man die Nachteile bestimmter Filterkonzepte, die vielleicht zu statisch arbeiten, erkennen, ohne aufwendige Firewall-Konzepte implementieren zu müssen.

Noch einmal aus implementierungstechnischer Sicht beschrieben: Ein Intranet besteht aus einem Objekt vom Typ `Net` und einem vom Typ `Router`. Hinzu kommen noch beliebig viele Objekte vom Typ `Server`. Dieses so gebildete Paket wird

von einer Anwendung (z.B. dem Simulations-Federate `Intranet`) initialisiert und getaktet. Jeder Server hat die NetzID seines Netzes. Die Simulation des gesamten Internets kann dann aus mehreren solcher Netzwerke bestehen.

Auf den einzelnen Servern kann man verschiedene Programme installieren, so dass sich verschiedene Server unterschiedlich verhalten können. Ebenso kann man auch das Verhalten des Routers manipulieren, indem man durch Verändern des Regelsatzes (bestehend aus Objekten des Typs `Rule`) sein Filterverhalten beeinflusst. Dieses Filterverhalten bestimmt, ob die zur Kommunikation unter Servern verwendeten Pakete (`Packet`) aus dem Netz hinausgelassen werden, bzw. ob Pakete aus anderen Netzen in dieses Netz hineingelassen werden. Der gesamte Aufbau ist in Abb. 21 dargestellt.

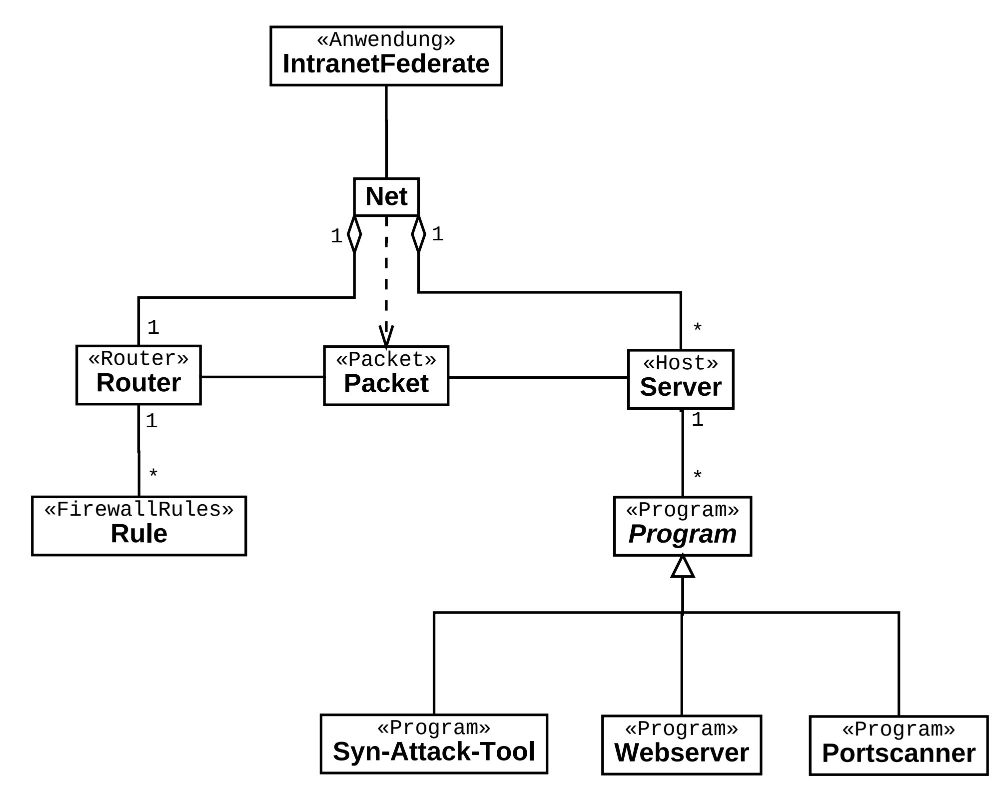
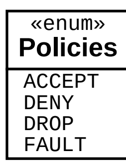


Abbildung 21: Aufbau des Intranets

Dieser Aufbau wurde so gewählt, um die Variationsmöglichkeit der Netzwerke zu gewährleisten. Man kann beispielsweise einfach den hier beschriebenen und verwendeten Router durch einen anderen ersetzen, der eine Firewall mit anderen Möglichkeiten der Filterung implementiert.

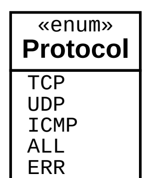
Im Folgenden wird nun auf die einzelnen Klassen des Netzwerks genauer eingegangen. Es wird ihr Aufbau beschrieben, und auf ihre Kommunikation sowohl untereinander als auch mit dem restlichen Internet eingegangen.

8.2.1 Basisklassen

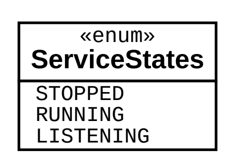


Für die bessere Übersichtlichkeit des Quellcodes und somit auch zur Vermeidung von Fehlern sind neue Basistypen definiert worden: Der Typ `Policies` definiert die möglichen Aktionen der Firewall, er legt die Namen der Aktionen fest. Es bleibt aber Sache des Routers, die dazugehörige Ausführung festzulegen. Dieser Basistyp wird von den Klassen `Router` und `Rule` benötigt. Implementiert wird dieser Typ als Aufzählung (C++: enum).

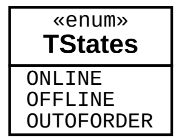
Der Basistyp `Protocol` ist ebenfalls eine Aufzählung und legt die Kommunikationsprotokolle fest, die in unserem Netzwerk zur Verfügung stehen.



Diese beiden Basisklassen haben neben den eigentlich verwendeten Aktionen bzw. Protokollen jeweils noch einen Fehlerzustand (`FAULT` bzw. `ERR`). Dieser ist nötig, um Fehler abzufangen, die beispielsweise beim Einlesen einer Firewallregel aus einem File auftreten können. Genauer wird auf dieses Problem im Kapitel über den Router (8.2.4) eingegangen.



`ServiceStates` ist ebenfalls ein Aufzählungstyp, der die Zustände definiert, die ein Programm annehmen kann. Denn der Server muss feststellen können, ob ein Programm gerade ausgeführt wird, ob es an einem Port lauscht, oder ob es nur installiert ist, aber nicht benutzt wird.



Abgeschlossen werden die Aufzählungstypen durch die Definition von `TStates`. Damit werden die möglichen Zustände definiert, die ein Server annehmen kann. Darunter sind die Normalzustände wie Online und Offline, aber auch der Zustand Out-of-order, der angibt, dass der Server auf irgendeine Art und Weise so ausgelastet ist, dass er normale Anfragen nicht mehr beantworten kann. Das kann durch eine SYN-Attacke ausgelöst werden, aber auch durch eine komplette Auslastung der Bandbreite. Wie hierbei zu erkennen ist, sind diese Zustände genau die, die im einfachen Opfermodell verwendet werden. Damit können wir auch ohne ein spezielles Opfer zu simulieren einen Angriff auf jeden Rechner der Netzwerke verfolgen.

Der letzte Basistyp `IP` ist eine richtige Klasse, die das in unserem Prototyp verwendete Äquivalent zu einer IP-Adresse definiert. Die IP-Adresse wird dabei als `int` gespeichert, wobei die letzten 8 Bit für die Hostadresse, die 8 Bit davor für die NetID verwendet werden. Dies gestattet unserem Prototyp theoretisch 256

Teilnetze zu simulieren, die jeweils bis zu 255 Hosts haben können. Die HostID 255 wird als Broadcast-Adresse verwendet.

```
IP
-wcNet: int
-wcHost: int
-ip: unsigned int
+IP(net:int,host:int)
+IP(id:char*)
+hostID(): int
+netID(): int
-setIP(net:int,host:int): void
-setAny(mask:int): void
+isAnyNet(): int
+isAnyHost(): int
+isEqual(other:IP*): int
```

Um einfacher auf HostID und NetzID der IP-Adresse zugreifen zu können, bietet `IP` die Funktionen `hostID()` und `netID()`, die das Gewünschte zurückliefern. Um zwei Adressen einfach vergleichen zu können, gibt es die Funktion `isEqual(IP)`, die überprüft, ob zwei IP-Adressen identisch sind. Identisch heißt hierbei: NetzID und HostID stimmen jeweils überein, entweder sind sie bei beiden IPs unbestimmt (Dargestellt wird dies mit "*") oder sie sind auf denselben Wert festgelegt. Anlegen kann man eine IP-Adresse über die Angabe von Netz- und HostID oder über die Angabe der IP-Adresse als String (Beispiel: "1.2"). Diese Schreibweise der IP-Adresse, die auch zum Ausgeben und Einlesen verwendet wird, ist der in wirklichen Netzkonfigurationen angepasst, um einfacher verständlich zu sein. Der große Unterschied ist nur, dass reale IP-Adressen aus 4 8Bit-Blöcken bestehen, während wir uns auf 2 8Bit-Blöcke beschränken. Das vereinfacht den Verwaltungsaufwand und scheint im Rahmen der Simulation vom Adressbereich her auszureichen. Sollte dieser Adressbereich irgendwann nicht ausreichen, kann er durch die Veränderung der Klasse `IP` erweitert werden.

Für die Implementierung z.B. einer Firewall ist es wichtig, auch IP-Bereiche angeben zu können. Auch diese Funktionalität bietet unsere Klasse `IP`. Der String "2.*" steht hier für alle IP-Adressen mit der NetzID 2. Ebenso sind auch Kombinationen möglich wie "*.*" für alle IP-Adressen bzw. "*.2" für die HostID 2 aus allen Teilnetzen. Mit den Funktionen `isAnyNet()` und `isAnyHost()` kann abgefragt werden, ob Netz- bzw. HostID auf einem bestimmten Wert n stehen oder unbestimmt sind ("*"). Um eine IP mit entsprechendem Platzhalter "*" zu erzeugen, kann auch wieder ein String verwendet werden, indem als Platzhalter für den unbelegten Wert "*" verwendet wird, oder man ruft den Konstruktor von `IP` nicht mit zwei Integern, sondern mit dem char "*" an der entsprechenden Stelle auf. Die genaue Syntax der Konstruktoren ist der Funktionsreferenz zu entnehmen (Kapitel B.1).

8.2.2 Klasse der Server

Die wichtigste Klasse in einem Netzwerk ist die Klasse der Server. Bevor wir uns mit den genauen Funktionen dieser Klasse befassen, erläutern wir zunächst ihren

Aufbau (siehe hierzu auch Abb. 22) sowie Funktionen, mit denen dieser Aufbau manipuliert werden kann.

Ein Server besitzt als Identifikation nach außen eine IP. Mit dieser ist er beim Netz bekannt, mit dieser markiert er legitime Pakete, damit der Absender erkennbar ist.

Um Pakete verschicken zu können, muss jeder Server eine Referenz auf sein Netz besitzen, denn über dieses Netz wird er seine Pakete verschicken. Auch wird er diesem Netz in jedem Zeitschritt eine Meldung über seinen eigenen Zustand schicken.

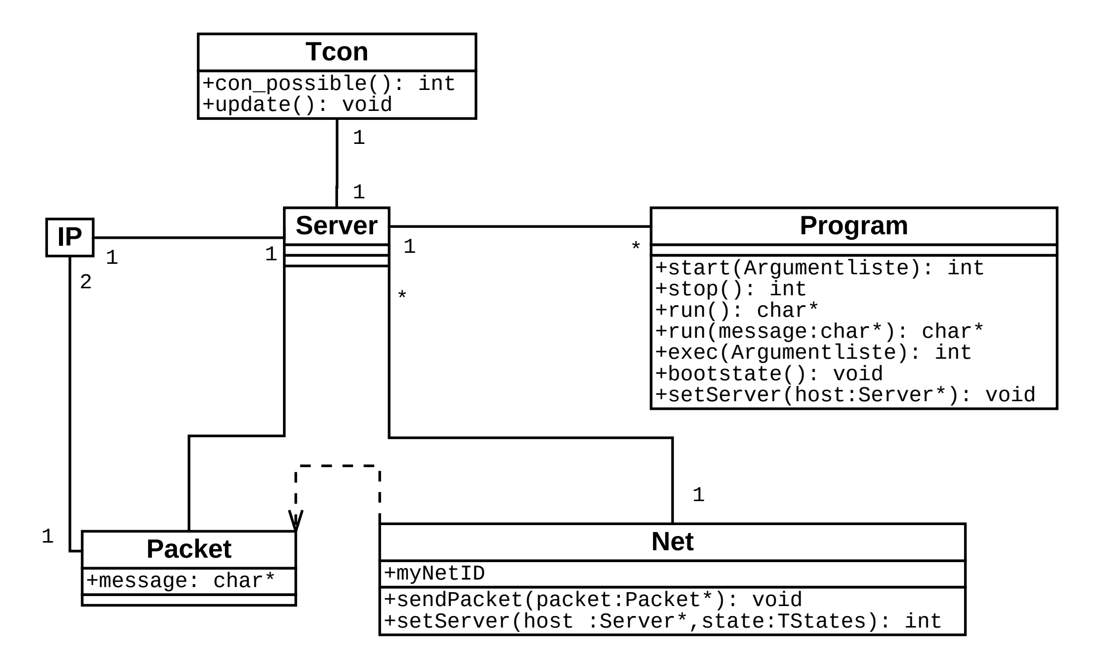


Abbildung 22: Das Umfeld des Servers

Der Server besitzt eine Liste, in der die auf ihm installierten Programme angegeben sind. Diese Liste kann von außen dynamisch beeinflusst werden, d.h. man kann Programme während der Simulation installieren und deinstallieren. Über diese Liste wird der Server auch die Programme zur Ausführung bringen, die während eines Zeitschrittes als laufend markiert sind.

Um ein Read-only-Filesystem zu simulieren, wird ein Flag namens `read_only` benutzt. Ist das Flag gesetzt, so ist das Filesystem read-only, sonst ist es schreibbar. Dieses Flag muss bei der Installation und Deinstallation von Programmen berücksichtigt werden. Weiter muss es eine Möglichkeit geben, dieses Flag zu verändern, um selbst bei einem read-only Filesystem das Verändern der

aktuellen Programmsituation des Servers zu erlauben. Wir simulieren somit das Filesystem des Servers als ein schreibbar oder read-only gemountetes Filesystem auf einem schreibbaren Speichermedium.

Weiterhin besitzt jeder Server ein Konfigurationsfile, in dem angegeben ist, welche Programme bei einer Neuinstallation auf diesem Rechner installiert werden. Dieses kann durch den Befehl `setConfig()` ersetzt werden oder bei einer neuen Installation durch Angabe eines alternativen Konfigurationsfiles ausgetauscht werden. Dabei wird nur der Name des Files als String samt Pfad angegeben. Dieser Pfad kann absolut angegeben werden oder auch relativ zu der Aufrufposition der Anwendung.

Neben den Programmen brauchen wir für die Kommunikation eine Möglichkeit, Programme an Ports zu binden, über die man die Programme dann ansprechen kann. Dazu existieren zwei Vektoren, deren Länge so festgelegt wird, dass der letzte ansprechbare Index der höchsten existierenden Portnummer entspricht. Dieser maximale Port ist durch einen festen Wert im Quellcode der Serverklasse bestimmt. Einer dieser zwei Vektoren ist die Portliste für TCP, der andere für UDP.

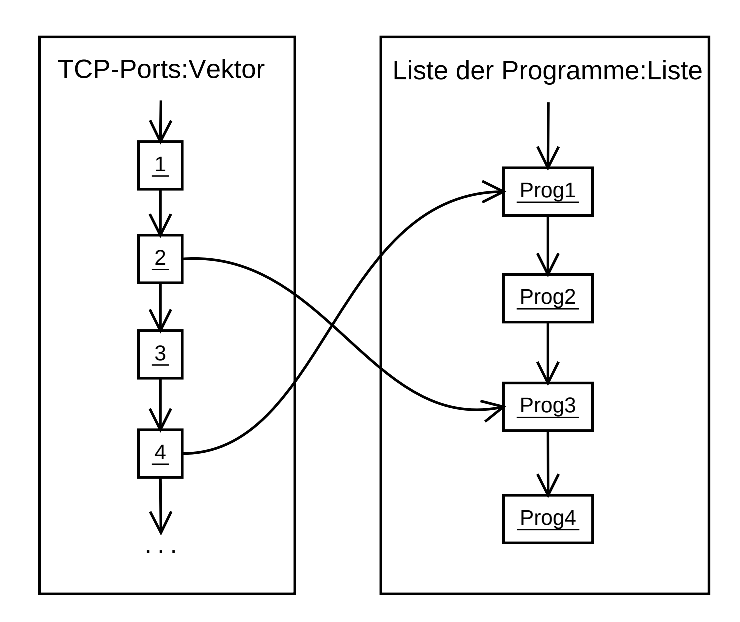


Abbildung 23: Programme werden auf Ports gelinkt

Um Programme an einen Port zu binden, wird eine Funktion namens `registerPort()` verwendet, der man die Adresse eines Programm-Objektes, die gewünschte Portnummer und das Protokoll mitgibt. So wird dann, falls der Port noch frei ist, von ihm ein Verweis auf das entsprechende Programm gesetzt. Das Programm ist dann durch den Port erreichbar. Ein Beispiel für eine solche Liste mit belegten Ports ist in Abbildung 23 zu sehen. Dort sieht man, dass das Programm

Prog1 an TCP-Port 4 gebunden wurde, und TCP-Port 2 ist an das Programm *Prog3* vergeben. Mit der Funktion `unregisterPort()` kann diese Bindung wieder gelöst werden.

Während es eine Liste von Paketen (wir nennen sie hier *toHandle*) gibt, die vom Server aktuell behandelt werden, werden eingehende Pakete zunächst in eine Warteschlange (*arrived*) eingereiht (bei Aufruf von `send()`), da sie erst im nächsten Zeitabschnitt bearbeitet werden dürfen. Zu einem Zeitwechsel wird dann die Liste der eingehenden Pakete an die (dann leere) *toHandle*-Liste angehängt, während die Eingangsliste (*arrived*) selbst geleert wird. Sie wartet dann wieder auf neue Pakete.

Die Pakete der *toHandle*-Liste werden der Reihe nach bearbeitet. Ist ein Paket bearbeitet worden, so wird es danach aus der Liste entfernt. Für jedes bearbeitete Paket wird ein Zähler erhöht, der angibt, wieviele Pakete zu der jeweiligen Zeit schon bearbeitet wurden. Da Server in der Realität auch nur eine bestimmte maximale Paketzahl pro Sekunde bearbeiten können, wird eine Konstante definiert, die für unsere Serverimplementierung ein solches Maximum angibt. Für unser Beispiel ist dieses auf den Wert 99 (`#define MAX_PACKETS 99`) gesetzt. Durch Änderung des `#define`-Macros in dem Include-File `server.h` kann dieser Maximal-Wert für die Paketverarbeitung verändert werden.

Hat ein Server dann zu einer Zeit die maximalen 99 Pakete bearbeitet, so darf er keine weiteren mehr bearbeiten. Er ist überlastet und entleert die *toHandle*-Liste, als hätte er die weiteren Pakete gar nicht mehr erhalten. Genaueres hierzu kann im Abschnitt 8.4 über die Implementierung der Angriffe nachgelesen werden.

Bei UDP wird bei der Bearbeitung der Inhalt der Pakete an das auf dem Port laufende Programm übergeben (Aufruf von `run()` mit den Daten als Parameter). Erzeugt dieser Programmlauf einen nicht-leeren String als Rückgabe, so wird dieser String in ein Antwortpaket gepackt und an den Sender des bearbeiteten Pakets zurückgeschickt. Läuft auf dem Port kein Programm, so wird ein ICMP-Port-unreachable-Paket generiert und an den Sender zurückgeschickt.

Bei UDP ist die Behandlung des Pakets damit abgeschlossen, bei TCP ist dies nicht ganz so einfach. Hier muss eine Datenstruktur eingeführt werden, mit der wir SYN-Attacken bearbeiten können und mit der wir vor dem Übermitteln der Daten an ein Programm überprüfen müssen, ob diese Aktion überhaupt durchgeführt werden darf. Ebenso muss anhand der gesetzten TCP-Flags ermittelt werden, welchem Zweck das Paket dient, und nur bei einer Möglichkeit wird der Paketinhalt an ein Programm übergeben.

Erklären wir nun die für die Bearbeitung der TCP-Verbindungsanfragen existierende Datenstruktur. Diese verwaltet die Anzahl der Anfragen, für die der TimeOut im gleichen Zeitschritt erfolgen wird, zusammen mit der Zeit, zu der sie in den

TimeOut laufen werden (Abb. 25). Hierzu wird eine Liste angelegt, deren Elemente (`Timenum`, Abb. 24) immer die Anzahl von Verbindungsanfragen und die dazugehörige Zeit, zu der der TimeOut erreicht wird, enthalten.

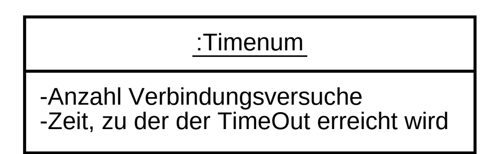


Abbildung 24: Ein Objekt der Struktur Timenum

Die Gesamtzahl der Verbindungsanfragen ergibt sich aus der Summe der in den Listenelementen stehenden Anzahlen. Auch für diese Gesamtzahl ist ein Maximalwert definiert, dessen Wert hier auf 10 (`#define MAX_CON 10`) gesetzt wurde. Eine Änderung dieses Wertes ist wieder im File `server.h` möglich.

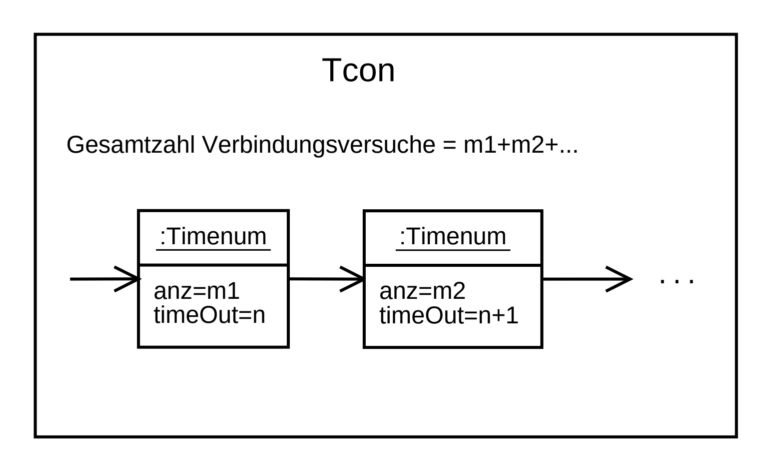


Abbildung 25: Illustration der Klasse Tcon

Gehen wir aber noch genauer auf die Datenstruktur `Tcon` ein. `Tcon` enthält eine Liste bestehend aus `Timenum`-Elementen. Diese Liste wird so behandelt, dass jedes Element eine andere Zeit enthält, wobei ein kleinerer Wert für die Zeit bedeutet, dass das Element weiter vorne in der Liste zu finden ist (Abb. 25).

Die Datenstruktur merkt sich auch die aktuelle Zeit. Diese bekommt sie bei dem Aufruf ihrer Funktion `update()` übergeben. Weiter wird während dieser `update()`-Funktion das erste Element der Liste darauf überprüft, ob die dort angegebene Zeit (`timeOut`) der aktuellen entspricht. Ist dies der Fall, so wird die Gesamtzahl der Verbindungsversuche um den Wert erniedrigt, der in dem Listenelement als Anzahl von Verbindungsversuchen angegeben ist. Dann wird dieses Listenelement entfernt. Ist das Element älter, so wird es genauso behandelt, wie

wenn die Zeit der aktuellen entspricht. Dieser Fall sollte zwar nicht auftreten können, da die Server zu jedem Zeittakt ihre Arbeit erledigen müssen und somit keine alten Listenelemente vorhanden sein sollten, da sie zum vorhergehenden Zeittakt hätten bearbeitet werden müssen. Und Zeitsprünge sind in dieser Implementierung nicht möglich.

Sollte dies aber doch aus irgendeinem Grund auftreten, so werden die Listenelemente von vorne solange überprüft, bearbeitet und entsorgt, bis das neue erste Element eine Zeit in der Zukunft angibt.

Weitere Funktionen von `Tcon` bieten die Möglichkeit, die aktuelle Gesamtzahl der Verbindungsversuche abzufragen (`getAnz()`) oder auch gezielt zu ermitteln, ob noch weitere Verbindungen aufgebaut werden können (`con_possible()`). Dazu muss die Bedingung erfüllt sein, dass die Anzahl der Verbindungsversuche kleiner ist als der definierte Maximalwert `MAX_CON`.

Mittels `reset()` kann die Liste geleert werden, dabei wird auch die Gesamtzahl der Verbindungsversuche auf 0 gesetzt. Dies ist z.B. bei einem Reboot des Servers nötig.

Bekommt ein Server ein Paket, mit dem ein Verbindungsaufbauversuch gemacht wird (z.B. ein Paket für eine SYN-Attacke), so wird die Funktion `inc()` von `Tcon` benutzt, um diesen richtig in die Liste einzusortieren und damit die Anzahl der Verbindungsversuche korrekt zu erhöhen. Dies wird aber nur gemacht, solange der Maximalwert nicht überschritten wurde, sonst hat die Funktion keine Wirkung.

Wird eine Verbindung in die Liste der Versuche aufgenommen, so wird sie dem Listenelement mit der eingetragenen Zeit $Zeit_{aktuell}+$ `CON_TIMEOUT` zugeordnet. `CON_TIMEOUT` ist dabei ein in `server.h` definierter Wert, der angibt, wielange ein Verbindungsaufbau dauert, bevor er durch einen TimeOut beendet wird. Existiert noch kein Listenelement mit dieser Zeit, so wird es neu erzeugt.

Mit den Kenntnissen, wie `Tcon` arbeitet, können wir nun genauer auf die Bearbeitung der TCP-Pakete eingehen. Wird ein TCP-Paket bearbeitet, so wird zunächst geprüft, ob der geforderte Port belegt ist. Wenn nein, wird ein ICMP-Port-unreachable an den Sender zurückgeschickt. Wenn ja, müssen wir das Paket weiter untersuchen.

Dazu wird geprüft, ob das SYN-Bit gesetzt ist. Wenn ja, müssen wir mit der `Tcon`-Funktion `con_possible()` abfragen, ob wir noch neue Verbindungen annehmen dürfen. Ist dies nicht der Fall, so liegt eine SYN-Attacke vor. Das Paket wird nicht weiter bearbeitet, das nächste Paket ist dran. Können wir das Paket noch annehmen, so unterscheiden wir: Ist das ACK-Bit nicht gesetzt, so ist dieses Paket Teil einer SYN-Attacke, wir rufen die `inc()`-Funktion unseres `Tcon`-Objektes

auf. Ist ACK gesetzt, so ist es ein ordentliches Paket. Anhand des FIN-Bits entscheiden wir, ob es ein Connect ist (FIN ist gesetzt), auf den wir dann eine Antwort generieren, oder ob wir eine Dienstausführung haben, und so die Daten an das Programm des gewünschten Ports übergeben und eventuell eine Antwort generieren.

Ist das SYN-Bit nicht gesetzt, gibt es drei Möglichkeiten, was für ein Paket uns erreicht: 1. Das ACK-Bit ist nicht gesetzt. Dieses Paket kann verworfen werden, da es eine ungültige Flag-Kombination besitzt. 2. ACK und FIN sind gesetzt. Dies ist eine Connect-Antwort. 3. ACK ist gesetzt, FIN nicht. Wir habe eine Antwort auf eine Dienstausführung erhalten und geben sie an das Programm weiter, welches an den Zielport des Antwortpakets gebunden ist. Bei Bedarf kann auf dieses Paket auch wieder eine Antwort generiert werden.

Dieser Ablauf ist in Abb. 26 dargestellt.

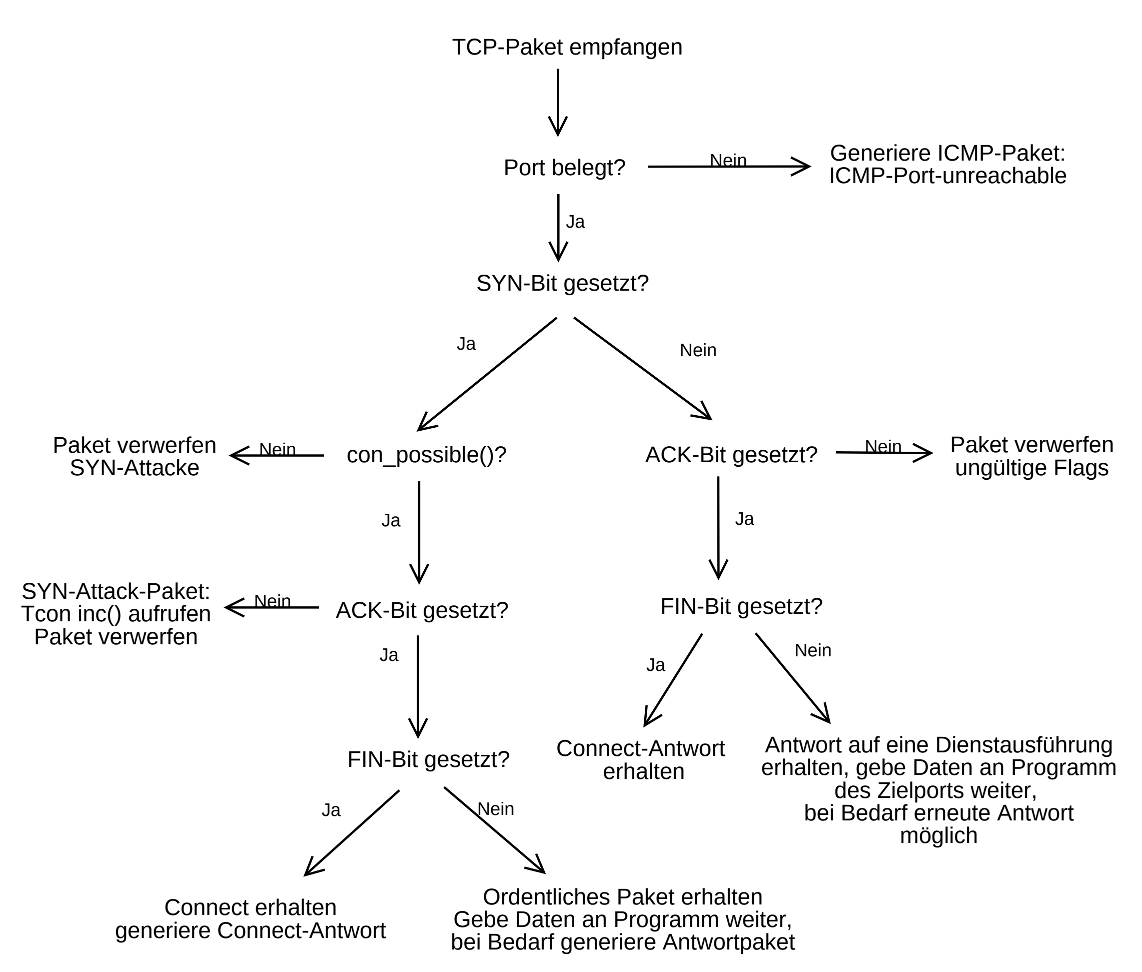


Abbildung 26: Ablauf der Bearbeitung eines TCP-Pakets

Jetzt wissen wir, wie TCP- und UDP-Pakete behandelt werden. Was passiert aber mit ICMP-Paketen? Da sie keine Portangaben besitzen, können wir sie nicht direkt an ein Programm weiterreichen. Außerdem werden z.B. Ping-Requests nicht von einem Programm bearbeitet, welches ein Ping-Reply generiert, sondern dies wird während der allgemeinen Bearbeitung der Pakete erledigt.

Wir implementieren dies auf folgende Art: Wird ein ICMP-Ping-Request bearbeitet, so generiert der Server ein Ping-Reply-Paket. Weiter wird jedes erhaltene ICMP-Paket in einer Liste für ICMP-Meldungen abgespeichert. So können Programme, die sich für ICMP-Meldungen interessieren, darauf zugreifen. Damit jedes Programm auch auf alle ICMP-Meldungen zugreifen kann, muss die Bearbeitung der Pakete vor der Ausführung der Programme stattfinden, denn zu jedem Zeitwechsel wird die ICMP-Liste wieder gelöscht. Sie würde sonst zu groß werden, und man bräuchte weiteren Aufwand, um die Pakete einer Zeit zuzuordnen, damit man nicht auf veraltete Pakete zugreift.

Eine Übersicht des Servers mit den bereits erwähnten Listen und Klassen- und Fileabhängigkeiten ist mit Abbildung 27 gegeben.

Wir haben nun mehrmals den Zeitwechsel erwähnt. Beschreiben wir genauer, wie dieser auf dem Server abläuft. Zuerst wird durch das Netz die Serverfunktion `newTime()` aufgerufen. Diese verschiebt, wie schon erwähnt, den Inhalt der Liste für ankommende Pakete in die Liste der zu bearbeitenden Pakete. Dadurch wird die Liste der ankommenden Pakete leer. Als Nächstes wird nun die Updatefunktion von`Tcon` aufgerufen, die die Verbindungsstatistik aktualisiert. Dann wird die ICMP-Liste geleert.

Der nächste Schritt des Zeitwechsels, der beim Server ankommt, ist die Ausführung der Funktion `doWork()`. Hier arbeitet der Server je nach seinem Zustand unterschiedlich. Ist er offline, so löscht er die Liste der zu bearbeitenden Pakete. Diese sollte zwar schon leer sein, aber wir wollen Fehler ausschließen. Dann wird eine Variable namens `stillDown` überprüft. Diese wird z.B. gesetzt, wenn eine Neuinstallation durchgeführt wird und wir für diese eine bestimmte Zeit veranschlagen, nach der der Server wieder erreichbar wird, also bootet. Ist diese Variable größer 0, so wird sie erniedrigt, der Server bleibt offline. Ist sie auf 0, so wird der Server gebootet, er ist wieder online.

Nun folgen die Aktionen, die ein Server, der online ist, durchführt (dazu gehört auch ein Server, der gerade eben wieder gebootet wurde). Der Server behandelt alle Pakete, die in der *toHandle*-Liste stehen (bis zur Maximalzahl, siehe oben). Danach wird die Liste der installierten Programme durchgegangen. Dort werden die Programme, die sich im Zustand `RUNNING` befinden, ausgeführt (`run()` wird aufgerufen). Genaueres zu den Programmen, ihrem Zustand, usw. siehe 8.2.3.

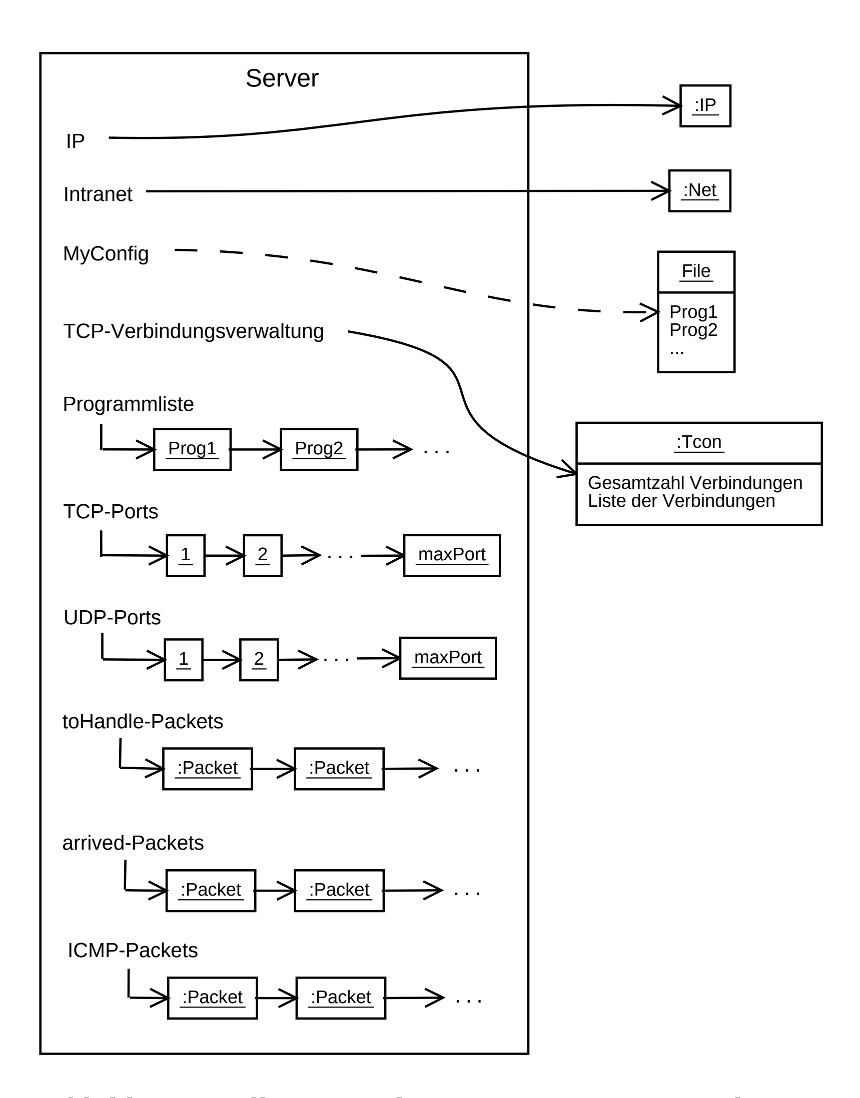


Abbildung 27: Illustration des Servers mit seinen Strukturen

Nachdem die Programme ausgeführt worden sind, ist die letzte Aufgabe des Servers, seinen Zustand an sein Netz und die RTI zu melden. Damit ist der Zeitwechsel abgeschlossen.

Beschreiben wir als Nächstes einen Bootvorgang, der durch die Funktion `boot()` durchgeführt wird. Hier sorgen wir zunächst dafür, dass die Programme des Servers nicht mehr laufen. Es wird bei sämtlichen Programmen die Funktion `stop()` aufgerufen, und danach `bootstate()`. Letztere Funktion führt die Programme in den definierten Zustand `STOPPED`. Die Programme sollten zwar schon nicht mehr laufen, aber auch hier wollen wir sicher gehen. Man kann sich nicht darauf verlassen, dass bei Erweiterungen des Prototyps z.B. um „Serverabstürze” der Zustand der Programme richtig gesetzt wird, und daher wollen wir bei der Im-

plementierung des Prototyps schon vorbeugen.

Dann wird dafür gesorgt, dass die Ports nicht an Programme gebunden sind. Danach werden die Programme gestartet, die zur Ausführung beim Bootvorgang durch die `Program`-Variable `startAtBoot` markiert sind. Der Server setzt seinen Zustand auf online.

Beim Shut-Down des Servers wird zuerst sein Zustand auf offline gesetzt, bevor weitere Aktivitäten zum Shut-Down ausgeführt werden. Diese bestehen dann aus Stoppen der Programme und Aufruf von `bootstate()`.

Eine Neuinstallation führt zuerst einen Shut-Down durch, danach wird die Programmliste geleert. Danach werden die im Konfigurationsfile angegebenen Programme installiert. Die Variable `stillDown` wird auf einen in `server.h` definierten Wert (hier 3) gesetzt. Damit soll der Server nach der in `stillDown` angegebenen Zeit wieder booten.

Um Programme auszuführen, können die Funktionen `execBefehl()` (Aufruf von `exec()` eines Programms) bzw. `startService()` verwendet werden. Von diesen Befehlen gibt es mehrere Versionen. Eine dient zur Ausführung ohne Parameter, der Server erzeugt dann selbst eine leere Parameterliste, die er an das Programm weitergibt. Die andere wird für den Aufruf mit einer Parameterliste verwendet. Weiter gibt es noch die Möglichkeit, den Befehl mit einer UserID oder ohne auszuführen. Die Version mit der UserID stammt aus dem Versuch, das Userkonzept (7.4) zu verwirklichen. Mit dieser Funktion wird der Befehl nur ausgeführt, wenn die angegebene UserID zu einem am System angemeldeten User gehört.[12] Die Funktion `stopService()` beendet die Ausführung eines als Dienst (mit `startService()`) aufgerufenen Programms.

Will ein Programm Netzwerkkommunikation betreiben, so stehen ihm mehrere Funktionen zur Verfügung. Mit `registerPort()` bindet sich das Programm an einen Port, mit `unregisterPort()` wird diese Verbindung gelöst. Um nun Pakete zu senden, sind die Funktionen `sendTo()` und `connect()` zu verwenden. Den `sendTo()`-Funktionen müssen die Zieldaten, Protokoll und Inhalt des Pakets angegeben werden, dann wird daraus ein entsprechendes Paket erzeugt. Wird explizit eine AbsendeIP angegeben, so wird diese als Absender eingesetzt, sonst wird vom Server die eigene, korrekte IP verwendet. Bei `connect()` werden keine Daten gesendet, es wird nur getestet, ob auf dem Zielport ein Programm arbeitet. Daher reichen dieser Funktion die Angaben von Absendeport, Ziel-IP und Zielport. Als Protokoll wird automatisch TCP eingesetzt, da diese Funktion bei den anderen Protokollen nicht existiert.

[12] Es gibt noch Funktionen zum Einloggen, Ausloggen, Passwortändern, usw. Genauer gehen wir hier nicht darauf ein, weil diese Funktionen in unserem Prototyp nicht mehr verwendet werden.

Die hier erwähnten Funktionen zur Paketerzeugung sorgen dafür, dass TCP-Pakete die Flags richtig gesetzt haben. Das aufrufende Programm braucht sich somit nicht um diese Flags zu kümmern.

Bei TCP und UDP wird bei der Bearbeitung der Inhalt der Pakete an das auf dem Port laufende Programm übergeben (Aufruf von `run()` mit den Daten als Parameter). Erzeugt dieser Programmlauf einen nicht-leeren String als Rückgabe, so wird dieser String in ein Antwortpaket gepackt und an den Sender des bearbeiteten Pakets zurückgeschickt.

Läuft auf dem Port kein Programm, so wird ein ICMP-Port-unreachable-Paket generiert und an den Sender zurückgeschickt.

8.2.3 Programme

Programme sind der aktive Teil eines Servers. Während der Server nur Kommunikationspakete erzeugen kann, um einen unbelegten Port zu melden, sind die Programme in der Lage, beliebig Pakete zur Kommunikation mit anderen Servern zu erstellen. Dies müssen sie können, wollen sie z.B. die Funktionalität eines Portscanners oder eines Tools für SYN-Attacken implementieren. Natürlich können die Programme auch rein serverinterne Aufgaben abarbeiten, wie z.B. ein Intrusion-Detection-Tool, welches überwacht, ob sich die Konfiguration des Servers verändert hat, und wenn dies der Fall ist, eine Neuinstallation anstößt. Aber es sind auch hier rein reaktive Aufgaben wie z.B. bei einem Webserver möglich.

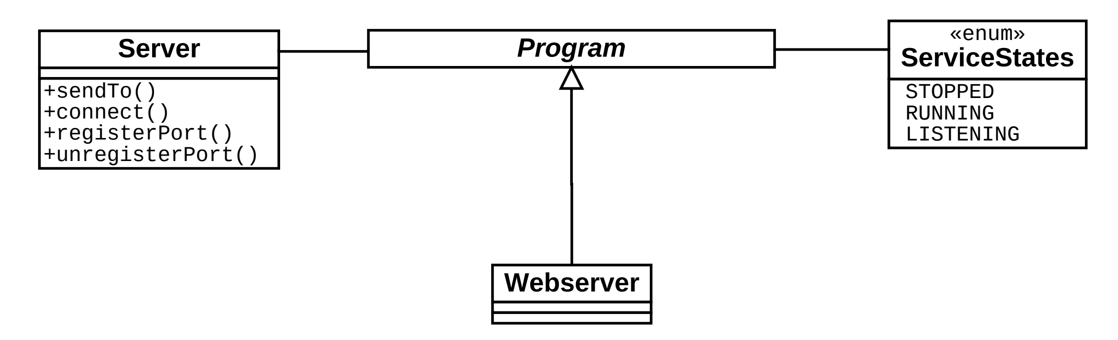


Abbildung 28: Das Umfeld von Programmen

Die Programme müssen sich immer in einem definierten Zustand (`ServiceStates`) befinden. Mögliche Zustände sind hier `STOPPED`, `RUNNING` oder `LISTENING`. `STOPPED` bedeutet, das Programm ist nicht am Arbeiten, es wird weder ausgeführt, noch ist es als Server durch Pakete von außen ansprechbar. Ein Programm im Zustand `LISTENING` ist an einem Port registriert und kann durch Senden eines Pakets an diesen Port zur Ausführung gebracht werden. Der

Zustand RUNNING deutet auf ein in Ausführung befindliches Programm hin. Nur ein Programm in diesem Zustand wird zu einem Zeitwechsel vom Server ausgeführt. Jedes Programm ist selbst dafür verantwortlich, einen konsistenten Zustand zu behalten. Der Server, auf dem das Programm läuft, kann dies nicht gewährleisten. Sonst bräuchte er Wissen über die Funktionalität der Programme, und dies ist nicht wünschenswert, weil es eine unrealistische Abbildung der Wirklichkeit wäre.

Einzig bei einem Reboot des Servers wird das Programm automatisch in einen definierten Boot-Zustand versetzt. So soll sichergestellt werden, dass bei einem Shut-Down des Servers alle laufenden Programme beendet werden. Hierzu stellt die Klasse Program eine Funktion bootstate() zur Verfügung, die das Programm in den Zustand STOPPED versetzt. Jedes Programm muss selbst darauf achten, dass es bei einem erneuten Start dann wieder seinen Zustand einem Programmstart anpasst, und nicht fälschlicherweise noch alte Daten berücksichtigt.

Alle Programme müssen gewisse Funktionen bereitstellen, damit der Server sie bedienen kann. Deshalb müssen die Programme, die man auf den Servern installieren können soll, das Interface Program (Abb. 29) implementieren. Dazu gehören Funktionen zum Starten (start()) und zum Stoppen (stop()) von Diensten. Im Falle eines Servers sollte z.B. das Programm bei start() auf einem bestimmten Port des Servers registriert werden, um es ansprechbar zu machen, und es sollte in den Zustand LISTENING gebracht werden. Denn ein Server kann nur so zur Ausführung gebracht werden. Bei stop() müsste der Port unregistriert werden und das Programm wieder den Zustand STOPPED erreichen. Aber auch andere dauerhaft laufende Programme, wie z.B. eines zur Eindringlingserkennung, können durch diese Funktionen gestartet und gestoppt werden. Sie sollten aber als Nicht-Serverdienst den Zustand RUNNING erhalten, um zu jedem Zeittakt vom Server ausgeführt zu werden. Ist ein Programm nicht für dauerhafte Ausführung gedacht, so sollten die Funktionen start() und stop() einfach nichts ausführen.

Die Funktion exec() ist für die Ausführung von Nicht-Diensten gedacht. Eine Exec-Ausführung sollte das Programm automatisch nach einiger Zeit wieder den Zustand STOPPED erreichen lassen, während ein von start()aufgerufenes Programm nur durch einen Reboot oder stop() wieder in diesen Zustand gelangt.

Man kann dieses System aber auch dazu missbrauchen, eine länger andauernde Ausführung des Programms zu simulieren. Dazu müssten in der Funktion exec() die Parameter zur Ausführung dauerhaft gesetzt werden, damit das Programm bei weiteren Aufrufen durch den Server noch Zugriff auf diese hat. Weiter muss man dann der Funktion run(), die die weiteren Aktionen durchführen muss, eine Abbruchbedingung (z.B. einen dekrementierenden Zähler) mitgeben, die durch Aufruf von exec() gesetzt wird.

Program

```
+name: char*
+startAtBoot: int
#myServer: Server*
+state: ServiceState
#myPort: int
#vulnerability: list<char*>
```

```
+start(argumente:list<char*>): int
+stop(): int
+run(): char*
+run(message:char*): char*
+run(fromIP:IP*,fromPort:IP*,message:char*): char*
+exec(argumente:list<char*>): int
+bootstate(): void
+setServer(host:Server*): void
+setVulnerabilities(): void
+setVulnerabilities(problem:char*): void
+patchVulnerabilities(problem:char*): void
```

Abbildung 29: Program-Interface

Obwohl dieses eigentlich nicht geplant war, stellte es sich doch als hilfreich heraus, mit diesem Trick zu arbeiten. So kann man beispielsweise durch Aufruf von `exec()` des Programms `Smurf` einen Angriff über mehrere Zeitschritte hinweg starten und muss nicht zu jedem Zeitschritt den Befehl wieder neu aufrufen.

Um Exploits anwenden zu können, braucht man aber auch eine Möglichkeit, dem Programm Schwachstellen anzugeben, die es besitzt. Hierzu gibt es eine Funktion `setVulnerabilities()`, mit der dem Programm eine von der Funktion fest vorgegebene Liste von Verwundbarkeiten angegeben wird. Dazu muss diese in `Program` virtuell definierte Funktion von jeder erbenden Programm-Klasse implementiert werden.

Um dann Sicherheitslücken durch Patches beheben zu können, kann die Funktion `patchVulnerabilities()` verwendet werden, mit der die durch einen bestimmten String identifizierte Schwachstelle behoben wird, sofern sie existiert. Da bei solchen Patches auch unter Umständen neue Schwachstellen eingeführt werden, oder im Laufe der Zeit neue Schwachstellen in Programmen entdeckt werden, existiert noch die Funktion `setVulnerabilities()`, mit der durch Angabe eines Strings als Identifikation einem Programm eine neue Verwundbarkeit hinzugefügt werden kann.

Die im Rahmen des Prototyps implementierten Klassen werden in Abschnitt 8.3 erläutert.

8.2.4 Router

Der Router ist für die Weiterleitung der Pakete aus dem Intranet ins Internet zuständig bzw. umgekehrt (Abb. 30). Ebenso hat er die Funktionalität einer einfachen Firewall, die Pakete nach Source-IP, Destination-IP, ZielPort, SourcePort

und Protokoll filtern kann. Als Aktion kann die Firewall ein Paket akzeptieren (`ACCEPT`), es fallen lassen (`DROP`) oder es zurückweisen, wobei dem Absender dann ein ICMP-Port-unreachable zurückgesendet wird (`DENY`). Siehe hierzu auch die UML-Sequenzdiagramme zum Routerverhalten (Abb. 31).

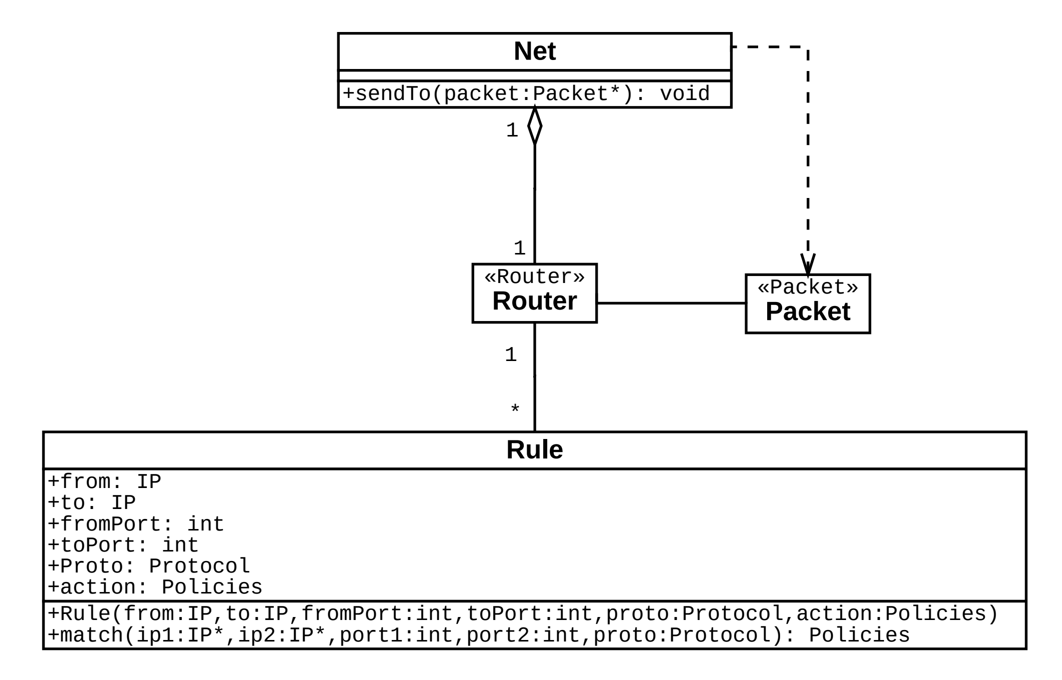


Abbildung 30: Das Umfeld des Routers

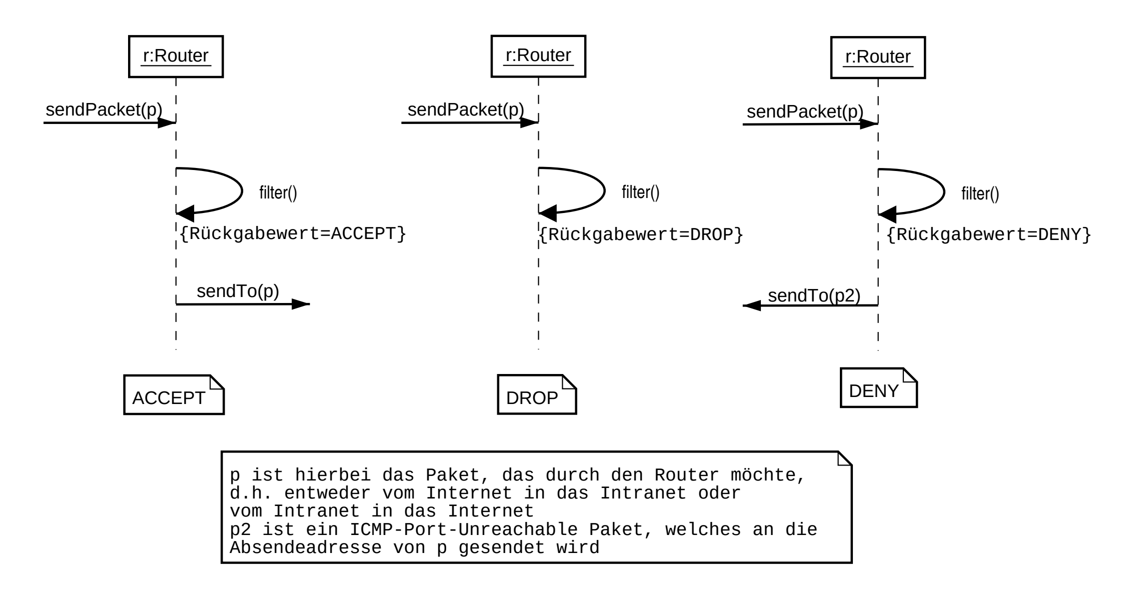


Abbildung 31: Verhalten des Routers bei verschiedenen Filteraktionen

Der Filter läuft den Regelsatz von oben nach unten durch, wobei die erste treffende Regel benutzt wird (Abb. 32). Trifft keine Regel zu, so wird die Standardregel (Default-Policy) benutzt (Abb. 33). Diese kann man mittels `setPolicy(aktion)` setzen und mit dem Befehl `getPolicy()` abfragen.

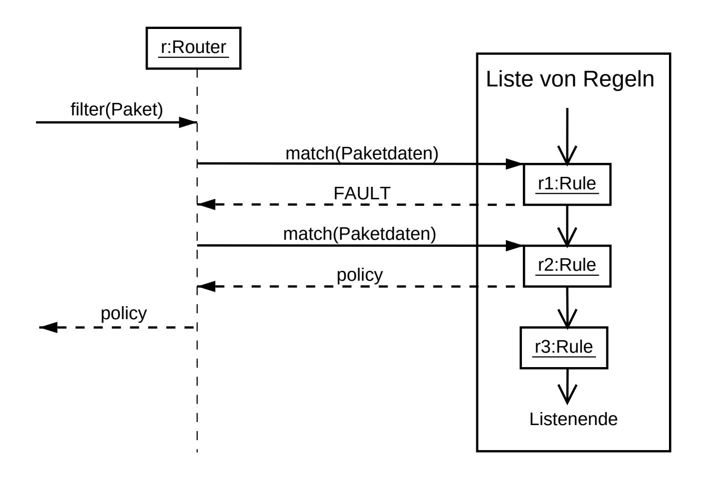


Abbildung 32: Suchen einer passenden Filterregel (erfolgreich)

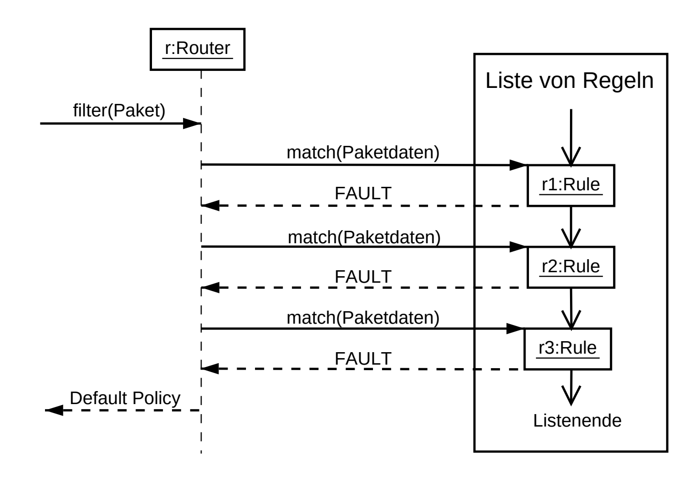


Abbildung 33: Suchen einer passenden Filterregel (schlägt fehl)

Der Regelsatz kann folgendermaßen verändert werden: Man kann einen Regelsatz aus einem File einlesen (`loadRuleset(filename)`), wobei auch die Default-Policy gesetzt wird, oder man kann einzelne Regeln zum bestehenden Regelsatz ergänzen (`addRule(*regel)`). Man kann den Regelsatz in ein File schreiben (`dumpRuleset(filename)`), welches man dann später auch wieder als Regelsatz laden kann, man kann es mit Zeilenzahl (`listRuleset()`) oder ohne (`dumpRuleset()`) ausgeben. Man kann einzelne Regeln löschen, indem man ihre Nummer angibt (`delRule(#regel)`), oder man kann alle Regeln löschen

(`clearRuleset()`). Ein Beispiel für ein Ruleset samt Erklärung ist Abb. 34 zu entnehmen.

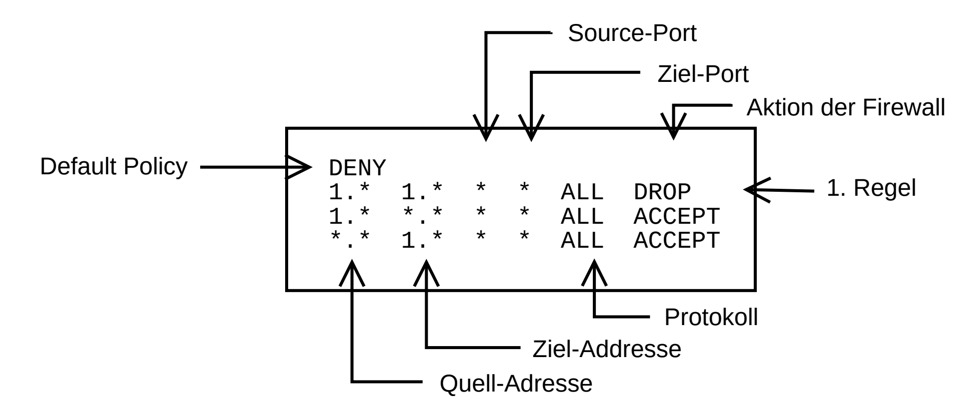


Abbildung 34: Aussehen eines Firewall-Regelsatzes

Da neu hinzugefügte Regeln automatisch an das Ende des Regelsatzes gehängt werden, die Reihenfolge der Regeln für die Wirkung der Firewall aber essentiell ist, muss man auch die Position der Regeln ändern können: Man kann eine Regel eine Position nach vorne (`mvRuleUp(#regel)`) oder nach hinten rücken (`mvRuleDown(#regel)`).

Die Klassendefinition zu `Router` ist in Abb. 35 zu sehen.

Router
-myNetID: int -fw: list<Rule*> -policy: Policies -intranet: Net* -packetList: list<Packet*>
+Router(myNet:Net*) +Router(myNet:Net*,rulefile:char*) +getPolicy(): Policies +setPolicy(pol:Policies): int +loadRuleset(rulefile:char*): int +clearRuleset(): int +listRuleset(): int +dumpRuleset(): int +dumpRuleset(dumpfile:char*): int +mvRuleUp(nr:int): int +mvRuleDown(nr:int): int +addRule(regel:Rule*): int +delRule(nr:int): int -filter(packet:Packet*): Policies +sendPacket(packet:Packet*): void -handlePacket(packet:Packet*): void

Abbildung 35: Klassendiagramm eines Routers

8.2.5 Netz

Das Netz dient als Steuereinheit für das Intranet. Es beinhaltet sowohl die Funktion, mit der das gesamte Netz in einem Zeitschritt seine routinemäßige Arbeit erledigt, als auch die Methode, die die „IP-Pakete” im Netzwerk zu ihrem Bestimmungsort leitet (siehe hierzu auch Abb. 36).

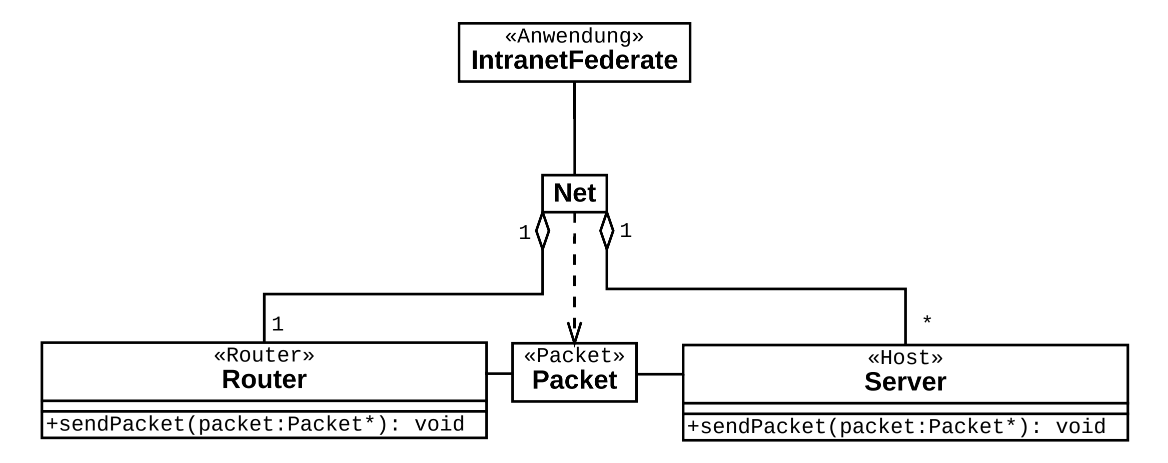


Abbildung 36: Umfeld des Netzes

Die Kommunikationspakete über so ein gesondertes Netz zu verschicken, hat gegenüber der direkten Kommunikation der Server untereinander einen großen Vorteil: Die Server müssen sich untereinander nicht kennen, sie müssen nicht wissen, ob der Bestimmungsort des Pakets wirklich existiert und wie er zu erreichen ist. Denn hierfür ist das Netz zuständig: Es kennt alle Server seines Netzes, ihren Zustand („online” oder „offline”) und kann so die Pakete ordnungsgemäß zustellen. Wenn ein Server nicht existiert, so kann es ein ICMP-Host-unreachable Paket zurücksenden, ansonsten wird das Paket beim Empfänger abgegeben bzw. beim Router, wenn der Bestimmungsort außerhalb des eigenen Netzes ist.

Für die zentrale Steuerung der Zeitverwaltung ist die Funktion `newTime(time)` zuständig, mit der die Taktung des Intranets zentral ausgeführt werden sollte. Dadurch wird sichergestellt, dass auch wirklich alle Server und Router des Netzwerks den Time-Advance mitbekommen und damit unter anderem alle Pakete, die im Zeitschritt vorher innerhalb des Intranets verschickt wurden, auch bearbeitet werden, und die Pakete, die in dieser Runde verschickt werden, auch wirklich erst im nächsten Zeitschritt bearbeitet werden und nicht zu früh. Weiterhin wird auf diesem Wege auch sichergestellt, dass alle auf den Servern laufenden Programme ihren Aufgaben nachkommen können (Siehe hierzu die Beschreibung der `doWork()`-Funktion der Klasse `Server`).

Dazu ruft das Netz zuerst von allen Servern die Funktion `newTime()` auf. Dabei regeln die Server ihre Paket-Queues. Der Router darf dann bei Aufruf seiner `newTime()`-Funktion neben der Umorganisation seiner Paket-Queues auch schon seine Pakete verschicken. Da alle sonst am Netz Beteiligten ihre Paket-Listen schon aktualisiert haben, kann keine Durchmischung neuer und alter Pakete mehr stattfinden. „Alte” Pakete sind hierbei Pakete, die den Server/Router im letzten Zeitschritt erreicht hatten und im aktuellen bearbeitet werden sollen, während „neue” Pakete diejenigen sind, die erst im aktuellen Zeitschritt bei den Servern/Routern ankommen. Danach dürfen auch die Server der Reihe nach ihre Pakete bearbeiten (dies geschieht in `handlePacket()`).

Der Ablauf der Funktion `newTime()` und seine Unteraufrufe an Server und Router ist in Abbildung 37 dargestellt.

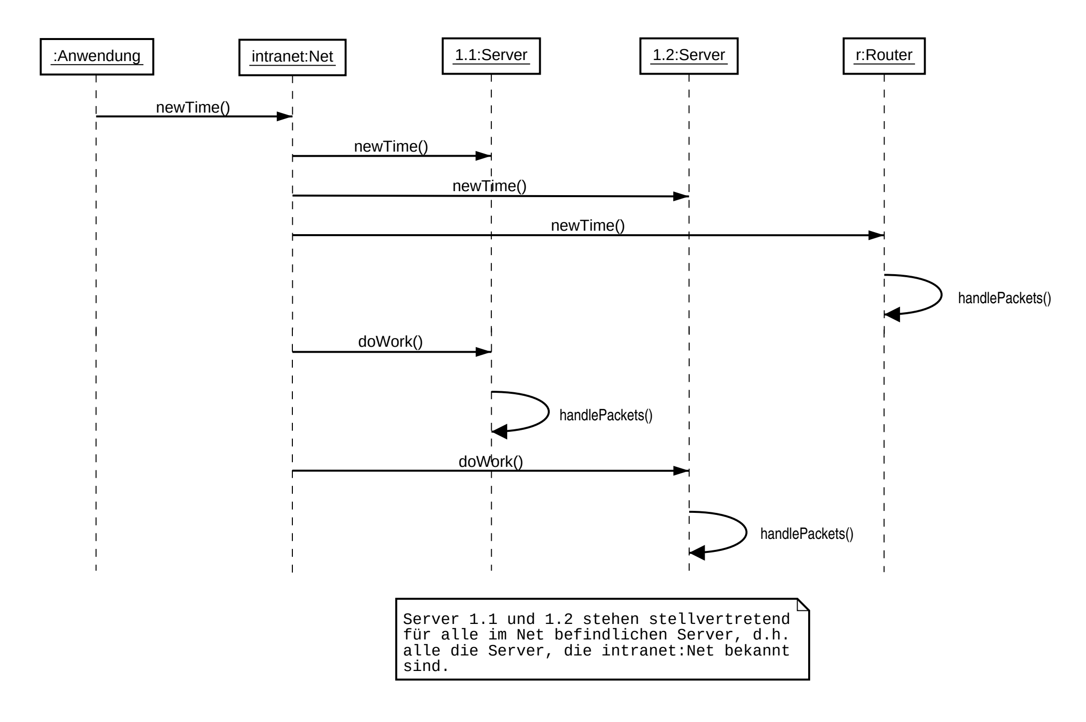


Abbildung 37: Ablauf eines Time-Updates im Netzwerk

8.2.6 Kommunikation im Netzwerk

Die Kommunikation in diesem simulierten Netzwerk ist genauso wie im realen Internet paketbasiert. Allerdings gibt es einen entscheidenden Unterschied: Während reale Protokolle dafür sorgen, dass größere Datenmengen in Teilpakete mit einer maximalen Größe zerlegt werden, kann unser Netzwerk jede Übertragung in einem

Paket durchführen. Dies führt zwar dazu, dass man spezielle Angriffe, die auf Dingen wie Paketfragmentierung beruhen (wie z.B. den berühmten „Ping of Death”), nicht simulieren kann, aber auch dafür kann man sich wenn nötig als Erweiterung des bestehenden Prototyps einen Ausweg ausdenken. Ein „Ping of Death” spielt in der heutigen Zeit zum Glück keine Rolle mehr, da die meisten Implementierungen der Betriebssysteme heute dagegen immun sind.

Weiterhin gibt es Entsprechungen der wichtigen Protokolle TCP, UDP und ICMP. Aber auch diese sind abstrahiert. Alle Protokolle arbeiten mit der selben Klasse `Packet`. Je nach Instanziierung der Attribute wird ein Paket als TCP-, UDP- oder ICMP-Paket interpretiert. Die möglichen Paketinstanziierungen mit kennzeichnenden Werten sind Abbildung 38 zu entnehmen. Die dort angegebenen Pakete sind noch in den Attributen für Source- und Ziel-IP, Source- und Zielport variabel. Ebenso ist der Datenanteil nicht festgelegt, außer durch die Anwendung, die die Pakete generiert.

Durch dieses Einheitspaket für alle Protokolle sind manche Attribute auch einmal unbelegt, oder sie beinhalten einen anderen Wert, als der Name andeutet. So ist bei ICMP und UDP das `flags`-Attribut unbelegt und sollte auch aus Protokollgründen nicht benutzt werden, um andere Daten damit zu codieren, da das bei den wirklichen Protokollen auch nicht möglich ist. Und die beiden Port-Attribute sind im Falle eines ICMP-Pakets nicht als Ports, sondern als ICMP-Typ und Code zu verstehen.

Ansonsten sind bei ICMP und UDP keine bemerkenswerten Änderungen vorgenommen worden. Aber das Protokoll von TCP ist um einiges vereinfacht worden. So entfällt das Bestätigen der empfangenen Pakete, da wir uns auf die Auslieferung der Pakete unter der RTI verlassen können und sich sonst deutliche Probleme mit dem Zeitverhalten der Kommunikation ergeben würden.

Weiterhin entfällt der eigentliche 3-Wege-Handshake. Dieser wurde aber symbolisch so nachgebildet, dass der SYN-Angriff dargestellt werden kann. So werden die für TCP bekannten Flags wie SYN, ACK, FIN und RST in dem Paket-Feld `flags` codiert. Für jedes TCP-Flag ist ein spezielles Bit von `flags` vorgesehen. Durch spezielle Kombination der TCP-Flags wird der Typ der TCP-Verbindung festgelegt. So ist bei jedem Paket, welches einen Verbindungsaufbau beinhalten muss, das SYN-Bit gesetzt. Wenn der Verbindungsaufbau korrekt ausgeführt werden kann, also der Absender ihn bei einem TCP-Handshake bestätigen würde, so wird auch das ACK-Bit gesetzt. Ein Paket eines SYN-Angriffs wird deshalb daran erkannt, dass es ein gesetztes SYN-, aber kein gesetztes ACK-Bit hat. FIN und RST spielen hierbei keine Rolle.

Will man keine wirkliche Kommunikation betreiben, sondern nur mit einem Connect testen, ob ein Port belegt ist, setzt man neben dem SYN und ACK noch das FIN-Bit.

Antwort-Pakete, also Pakete, die eine bestehende Verbindung nutzen, können nur im direkten Anschluss an eine Übermittlung der Gegenseite stattfinden, da sich die Server bestehende Verbindungen nicht merken. Wenn z.B. ein Server durch ein Paket einen Dienst ausgeführt hat, dann kann direkt im Anschluss dieser Ausführung ein Paket als Antwort gesendet werden. Dabei sind die Flag-Belegungen wie bei den verbindungsaufbauenden Paketen, nur dass das SYN-Bit ungesetzt bleibt. Eine Antwort auf ein Connect hat also ACK und FIN gesetzt, wenn ein belegter Port bestätigt wird, während eine normale Antwortübermittlung eines Dienstes nur ein gesetztes ACK-Bit hat.

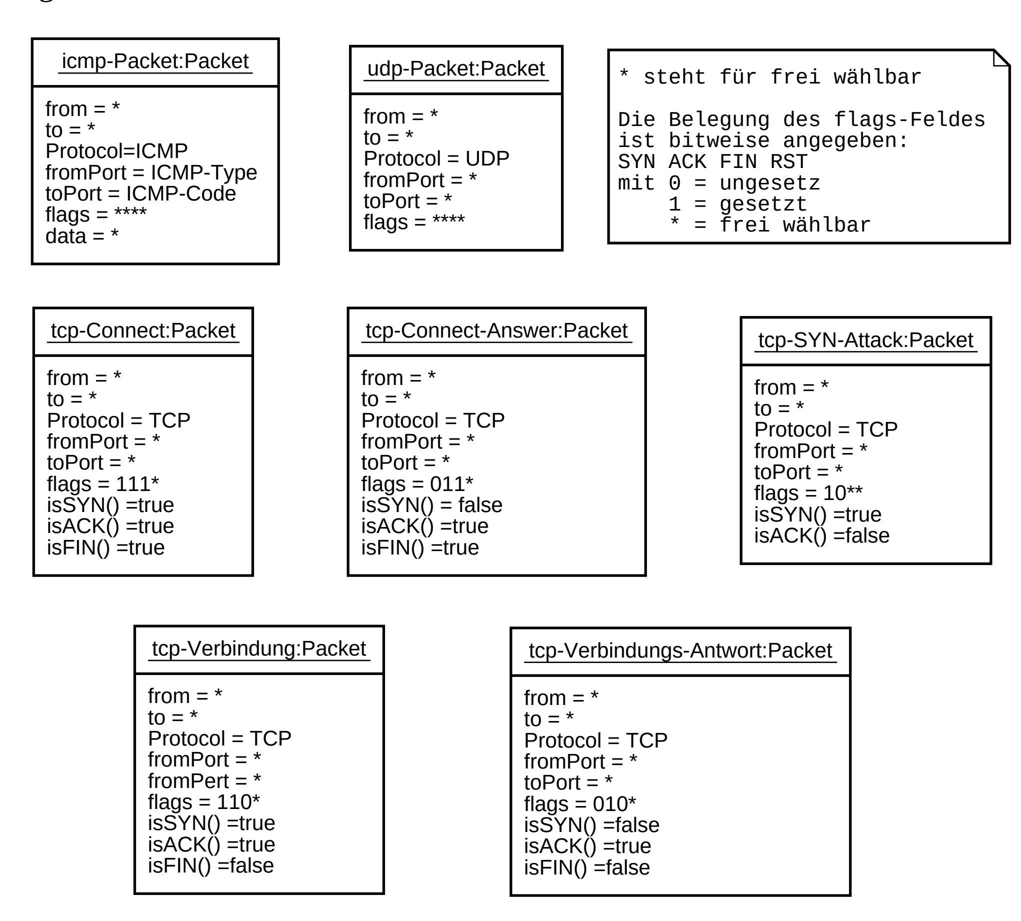


Abbildung 38: Übersicht über die Pakettypen

In Abb. 39 ist der Ablauf eines Connects im Intranet dargestellt. Dabei erzeugt der Server ein Paket, welches er an sein Netz weitergibt. Im Beispiel stellt das Netz fest, dass das Paket lokal verschickt wird, und gibt es an den gewünschten Empfänger weiter. Dieser reagiert dann seiner Konfiguration und Lage entsprechend auf den Empfang: Wenn der Port, den das Paket ansprechen will, belegt ist, dann erzeugt er ein Antwortpaket, welches den Connect bestätigt, und sendet es an den In-

itiator des Connects zurück (Abb. 39 (a)). Ist der Port nicht belegt, so wird statt der Connect-Antwort ein ICMP-Port-unreachable-Paket generiert und zurückgeschickt (Abb. 39 (b)).

Das ist das gewünschte Verhalten eines Connects. Aber stellt man sich den Fall einer laufenden SYN-Attacke vor, so darf der Server den Connect nur beantworten, wenn er die nötigen Ressourcen dazu noch frei hat. Wenn der Server feststellt, dass der Port belegt ist, so muss er noch zusätzlich abfragen, ob er noch Verbindungen annehmen kann. Und nur dann darf er die Connect-Antwort senden. Wenn keine Ressourcen frei sind, so verhält er sich, als hätte er das Paket nicht bekommen: Es findet keine Antwort statt (Abb. 39(c)).

Der Unterschied zwischen der Kommunikation intranet-intern und der Kommunikation von Servern in unterschiedlichen Netzen ist nicht groß. Der Server im Sendenetz generiert hier genauso ein Paket und sendet es an sein Netz. Dieses stellt dann fest, dass das Paket in ein fremdes Netz geschickt werden soll, und reicht es an den Router weiter. Der Router überprüft, ob das Paket das Netz verlassen darf und sendet es dann an das Internet, welches durch die RTI repräsentiert wird. Der Router initiiert also das Senden des Pakets als HLA-Interaktion an die RTI. Diese sorgt dafür, dass die Subnetze von diesem Paket erfahren. Nur das gewünschte Empfangsnetz übernimmt dann die Weiterbearbeitung des Pakets, alle anderen Netze ignorieren es. Der Router des Empfangsnetzes überprüft, ob das Paket nach seinen Filterregeln in das interne Netz weitergereicht werden kann und tut dies gegebenenfalls. Das interne Empfangsnetz gibt dann das Paket an den Empfangsserver weiter, der mit dem Paket ebenso verfährt, wie der Empfänger es bei rein intranet-interner Kommunikation getan hat. Das eventuell generierte Rückpaket wird auf dieselbe Art an das Sendenetz geschickt, wie vorher das Sendenetz das Paket an das Empfangsnetz gesendet hat.

8.2.7 RTI-Anbindung des Netzwerks

Bisher wurde nur erwähnt, dass Server/Router Pakete oder Meldungen an die RTI schicken. Wie geschieht dies aber genau? Und auf welchem Weg erfährt das Intranet davon, dass andere Netze Pakete an es geschickt haben?

Die Interaktion mit der RTI läuft über das Anwendungsprogramm `Intranet` und zwei weitere Klassen, den FederateAmbassador `IntranetFedAmb` und den `RTIhandler`, der die RTI-Funktionen für andere Klassen versteckt. Die im Anwendungsprogramm und dem `RTIhandler` erwähnten Funktionen werden von der RTI bereitgestellt und über den RTI-Ambassador aufgerufen.

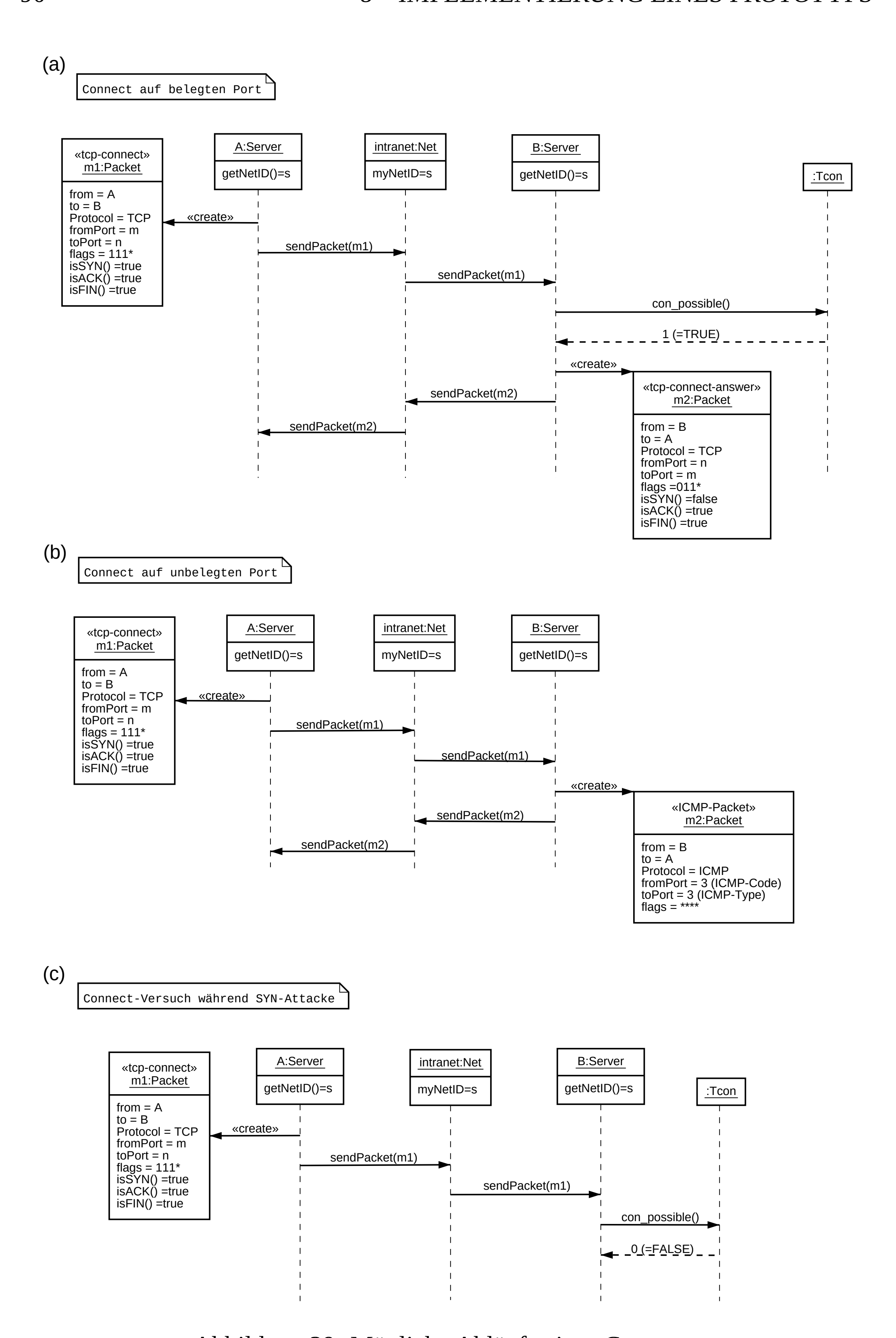


Abbildung 39: Mögliche Abläufe eines Connects

RTI-Aktionen des Anwendungsprogramms

Der Federate muss sich zunächst bei einer Federation-Execution anmelden. Diese hat im Prototyp den Namen DoS-Sim. Damit diese Execution auch existiert, versucht jeder Federate zunächst, sie mit `createFederationExecution()` neu einzurichten. Entweder sie ist schon vorhanden, oder sie wird neu angelegt, beide Fälle sind zulässig und hindern den Federate nicht an weiterem Vorgehen. Danach meldet er sich mit `joinFederationExecution()` bei ihr an.

Diese Aktionen werden alle vom Anwendungsprogramm `Intranet` übernommen, ebenso wie die Ansage an die RTI, über welche Interaktionen der Federate informiert werden möchte. Trifft eine solche Interaktion ein, so wird dies dem FederateAmbassador `IntranetFedAmb` gemeldet, der der Ansprechpartner des Federates für die RTI ist.

Die Interaktionen, die das Intranet an die RTI schicken möchte, werden im `RTIhandler` an die RTI gemeldet. Dort werden alle für die Statusmeldungen und das Verschicken von Paketen an das Internet nötigen Interaktionen behandelt. Da das Intranet sonst keine weiteren Interaktionen zu verschicken braucht, muss die Anwendungsklasse selbst keine Interaktionen anmelden.

Aber Interaktionen, die man von anderen erhalten möchte, müssen bei der RTI gemeldet werden. Deshalb meldet das Programm nun mittels `subscribeInteractionClass()` an, dass es die Interaktionen IPpacket, SimulationEnd, AttackerResign und AttackerJoin erhalten möchte.

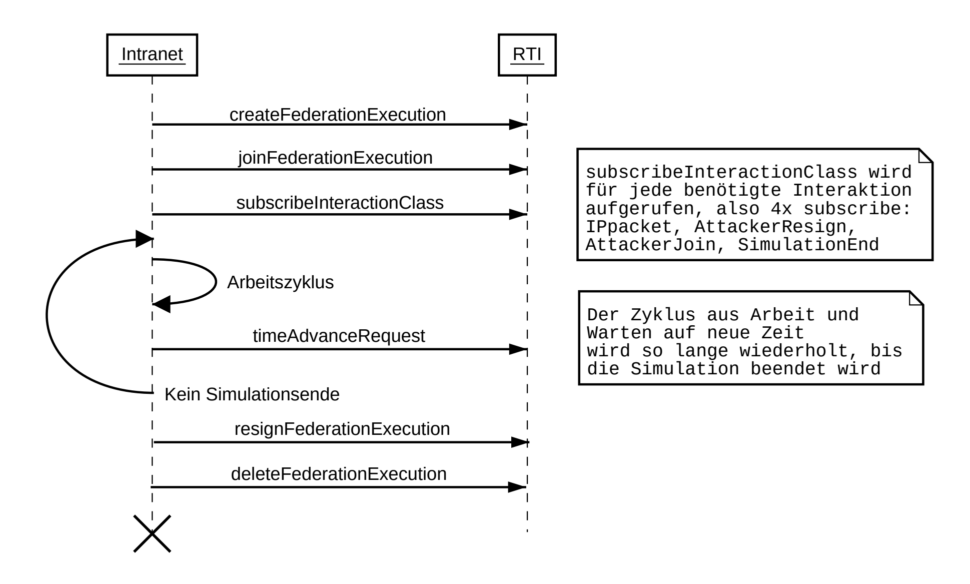


Abbildung 40: RTI-Aufrufe aus der Anwendung Intranet

Bevor die eigentliche Arbeit des Netzes beginnt, setzt das Programm noch sein Timing-Verhalten so, dass es sowohl zeitbestimmend, als auch zeitabhängig ist. Dies geschieht mit den Funktionen `enableTimeRegulation()` und `enableTimeConstrained()`.

Die Federates, die Netzwerke simulieren, sollten diese beiden Timing-Verhalten besitzen, da es sonst Probleme mit der Kommunikation der Federates geben kann. Durch die Eigenschaft des Zeitbestimmens müssen alle anderen Federates warten, bis auch dieser Federate mit den Aktionen eines Zeitabschnitts fertig ist, bevor sie in der Zeit voranschreiten dürfen. Dadurch kann kein anderer Federate schon den nächsten Zeitschritt ausführen, solange dieser Federate noch RTI-Interaktionen absenden kann. Durch die Zeitabhängigkeit kann aber auch dieser Federate noch nicht in die Zukunft voranschreiten, solange für ihn noch Interaktionen und andere RTI-Meldungen eintreffen können.

Durch dieses Zeitverhalten und die Einhaltung von Regeln bei der Versendung von Meldungen an die RTI wird verhindert, dass ein Federate in der Zeit den anderen vorauseilt. Dieses Verhalten nennt man bei verteilten Simulationen „konservativ". Ein anderes mögliches Verhalten wäre, dass jeder Federate möglichst schnell in der Zeit voranschreitet und dann bei Ereignissen, die in der „Vergangenheit" auftreten, diese entweder ignoriert, oder seine Ausführung abbricht und ab dem Zeitpunkt, zu dem das Event eingetroffen ist, wiederholt. Für unseren Zweck scheint die konservative Vorgehensweise aber die sinnvollste zu sein.

Die Anwendung wartet auf die Bestätigung, dass die RTI für sie die Zeiteigenschaften gesetzt hat, dann kann sie mit ihrer eigentlichen Arbeit beginnen. Diese besteht dann aus dem Zyklus *Arbeit eines Zeitabschnittes erledigen* und dann *Warten auf den nächsten Zeitabschnitt*. Um zum nächsten Zeitabschnitt voranschreiten zu dürfen, muss jeder Federate erst um die Erlaubnis dafür bei der RTI bitten. Dies geschieht mittels des Aufrufs von `timeAdvanceRequest()`.

Soll die Simulation beendet werden, so muss `Intranet` seine Teilnahme an der Federation-Execution beenden. Dies geschieht über `resignFederationExecution()`. Danach versucht der Federate die Execution zu beenden (`deleteFederationExecution()`) und beendet sein Programm. Die Execution wird dann beendet, wenn der letzte Federate die Ausführung verlassen hat und jemand versucht, sie zu löschen.

IntranetFedAmb

Die im Folgenden erwähnten Variablen zur Mitteilung von Sachverhalten zwischen dem Hauptprogramm `Intranet` und dem Federate Ambassador `IntranetFedAmb` sind global in `Intranet` definiert und über `export` in den

IntranetFedAmb eingebunden. Das ist zugegebenermaßen kein guter Stil, wurde aber aus Beispielprogrammen zur RTI-Anwendung übernommen und aus Mangel an Zeit nicht mehr geändert.

Der IntranetFedAmb erhält über die von ihm implementierte Funktion receiveInteraction() von der RTI Meldungen über eingehende Interaktionen, aber nur über die, die sein Federate per subscribeInteractionClass() bei der RTI abonniert hat, also IPpacket, SimulationEnd, AttackerResign und AttackerJoin.

Die Interaktion IPpacket steht für das Eintreffen eines Pakets. Der IntranetFedAmb überprüft, ob es die zu ihm gehörige NetzID aufweist. Wenn ja, so wandelt er die durch die Interaktion empfangenen Daten in ein Objekt vom Typ Packet um und sendet es an den Router, wenn nein, ist das Paket für ein anderes Netz, und die Interaktion wird nicht weiter ausgewertet. Um seine Netz-ID herauszufinden, kann der IntranetFedAmb die Funktion getNetID() seiner Net-Klasse benutzen.

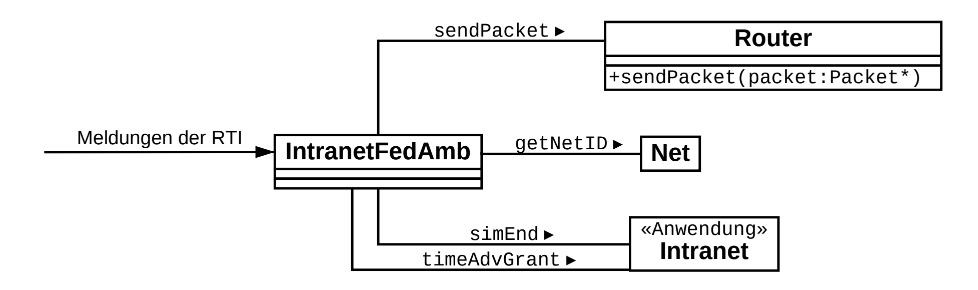


Abbildung 41: Einbindung des Intranet-Federate-Ambassadors

SimulationEnd unterrichtet den Federate, dass die Simulation beendet werden soll. Der IntranetFedAmb setzt den Wert der Variablen simEnd auf 1 und teilt so dem Anwendungsprogramm mit, dass es sich von der Execution abmelden soll.

Erreicht die Interaktion AttackerJoin den Federate, so erhöht er einen Zähler. Bei Eingang der AttackerResign-Interaktion wird der Zähler wieder verkleinert. Dieser Zähler gibt also die Anzahl der aktuell bekannten Angreifer an, oder genauer die Zahl der eingegangenen AttackerJoin-Interaktionen minus der Anzahl der eingegangenen AttackerResign-Interaktionen. Erreicht der Zähler beim Erniedrigen den Wert 0, so wird die Simulation beendet, da der letzte Angreifer aus der Simulation ausgestiegen ist.

Die Klasse IntranetFedAmb ist auch für den Empfang der Bestätigungen für Zeitverhalten und Voranschreiten in die neue Zeit zuständig. Über die Funktionen enableTimeConstrained() bzw. enableTimeRegulation() erhält der

Federate die Bestätigung, dass die RTI diese Verhaltensweisen für ihn anwendet. Damit das Hauptprogramm dies erfährt, setzt der Federate-Ambassador die Variablen `TimeRegulation` und `TimeConstrained` bei Erhalt der jeweiligen Bestätigung auf den Wert 1.

Das Voranschreiten in der Zeit wird über die Federate-Ambassador-Funktion `timeAdvanceGrant()` verwaltet. Diese setzt die Variable `grantTime` auf die von der RTI zugewiesene neue Zeit. Auch hier wird das Erreichen der Meldung über das Setzen der Variablen (`timeAdvGrant`) dem Hauptprogramm mitgeteilt.

Die anderen Funktionen, die bei der Beschreibung von HLA für den Federate-Ambassador angegeben wurden, sind für das Intranet nicht von Bedeutung. Ihre Funktionalität wird nicht gebraucht, also sollten sie von der RTI auch nicht aufgerufen werden. Falls doch, wird vom Federate-Ambassador eine Fehlermeldung generiert.

RTIhandler

Der `RTIhandler` ist für die RTI-Aufrufe der Klassen aus dem Netzwerk zuständig. Dazu gehören das Verschicken der Pakete vom Router an das Internet oder die Statusmeldungen, die die Server über ihren Zustand geben (Abb. 42).

Bei Erzeugen des `RTIhandlers` werden die Interaktionen, die der Handler verschicken können soll, bei der RTI zur Veröffentlichung angegeben (`publishInteractionClass()`). Hierzu gehören die Interaktionen IPpacket, ServerStatus und ServerNewInstall.

Die Interaktion IPpacket wird verwendet, um die Pakete der Netzwerkkommunikation über die RTI zu versenden. Dazu wird vom Router die Handler-Funktion `sendPacket()` aufgerufen. Mehr als dass man mit dieser Funktion Pakete an die RTI sendet, braucht der Benutzer von `Router` oder der Autor einer neuer Klasse mit Routerfunktion nicht zu wissen.

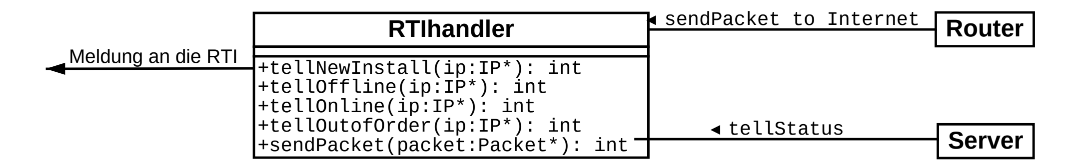


Abbildung 42: Der RTIhandler wird von Server und Router gebraucht

Die beiden anderen Interaktionen geben den Status der Netzwerkserver an die RTI weiter. Dazu werden von den Servern die Handler-Funktionen

tellOnline(), tellOffline() oder tellOutofOrder() benutzt, um eine Interaktion der Klasse ServerStatus zu versenden. Diese hat als Inhalt die IP des aufrufenden Servers (angegeben als Parameter der Funktion) sowie seinen Zustand (von Typ TStates) als String. Die Interaktion ServerNewInstall, die eine Neu-Installation eines Servers meldet und dabei die IP des neu installierten Servers als Mitteilung an die RTI schickt, wird durch den Aufruf der Handler-Funktion tellNewInstall() eingeleitet.

8.3 Implementierte Programme

Es wurden zu Test- und Simulationszwecken einige Programmklassen geschrieben, die nun erläutert werden. Diese Klassen sind alle Erben von Program.

8.3.1 IDT

Die Klasse IDT dient als Beispielimplementierung für ein Intrusion-Detection-Tool.

Beim ersten Start des Programms durch start() wird eine Referenzliste mit den auf dem Server laufenden Programmen erzeugt und gespeichert. Bei jedem weiteren Aufruf von start() wird die bestehende Liste weiterverwendet, vorausgesetzt sie ist nicht leer. Denn dann würde das IDT ungefragt eine neue Liste generieren.

Bei run() wird die Liste der auf dem Server installierten Programme mit der Referenzliste verglichen. Tritt eine Abweichung auf, so regt das IDT eine Neuinstallation des Servers an. Sind die Listen identisch, so wird nichts unternommen.

Um bei einer beabsichtigten Installation eines neuen Programms das Tool nicht Alarm auslösen zu lassen, gibt es eine Update-Funktion für das Tool: Der Aufruf von exec() mit dem String ”update” als erstes Element der Argumentliste sorgt für eine Aktualisierung der Referenzliste.

8.3.2 Ping

Das Ping-Programm sendet bei Aufruf durch exec() an jede IP, die in der Argumentliste steht, ein ICMP-Ping-request-Paket. Dieses Programm dient zum Test der Verbindung im Prototyp. So z.B. bei der Simulation in Kapitel 9, wo damit gezeigt wird, dass nicht alle Pakete durch die Firewall geblockt werden. Weiter dient dieses Programm zum Test der ICMP-Implementierung der Server-Klasse.

8.3.3 Smurf

Dies ist die Klasse, mit der ein Smurf-Angriff ausgeführt werden kann. Sie sendet bei `exec()` einmalig 10 Ping-Pakete an die Adresse, die an zweiter Position der Argumentliste steht. Das erste Listenelement gibt die Adresse des Opfers an, also die Absendeadresse, die in den Ping-Paketen genannt wird. Diese Adressen werden als C-String angegeben, also beispielsweise als ”1.1”. Enthält die Argumentliste weniger als zwei Elemente, so bricht das Programm die Ausführung ab, weitere Argumente als die zwei benötigten werden ignoriert.

Neben dieser auf einen Zeitschritt beschränkten Smurf-Attacke kann man aber auch die Ausführung des Angriffs durch `start()` beginnen. Der Angriff läuft dann wie oben beschrieben, nur eben so lange, bis `stop()` aufgerufen wird. Zu dieser Ausführung wird die bei `start()` übergebene Argumentliste in eine dauerhafte umgesetzt, die auch bei Aufruf von `run()` ausgelesen werden kann.

8.3.4 SynAttackTool

Das `SynAttackTool` dient zum Test des SYN-Angriffs und damit der `Server`-Implementierung für TCP-Verbindungen. Wird der Befehl `exec()` aufgerufen, so startet es eine SYN-Attacke, die über 10 Zeitschritte läuft. Die SYN-Attacke versucht, den Port 8 auf Opferseite für die Attacke zu nutzten. Dieses ist der Standardport für Webserver in unserem Szenario. `exec()` setzt auch den Status des Programms auf `RUNNING`, damit das Programm auch in den folgenden Zeitschritten ausgeführt wird. Ein Zähler *turns* wird auf den Wert gesetzt, wie lange das Programm laufen soll (hier 10). Solange das `SynAttackTool` im Zustand `RUNNING` ist, wird es jeden Zeittakt vom Server aufgerufen und führt dabei weiter seine Attacke durch. Jeden Takt wird dabei der Zähler `turns` erniedrigt und wenn er 0 erreicht hat, beendet sich das Programm, es nimmt den Zustand `STOPPED` an. Alternativ kann man die Ausführung auch vorzeitig mit dem Befehl `stop()` abbrechen. Auch dann wird `turns` auf 0 gesetzt und das Programm geht in den Zustand `STOPPED` über.

Während des Angriffs werden pro Zeittakt je 15 TCP-Pakete an das Opfer gesendet, das als Argument des `exec()`-Befehls angegeben wurde. Diese 15 Pakete besitzen unterschiedliche Absende-IPs.

8.3.5 `Webserver`

Der Webserver wird mit dem Befehl `start()` in den Zustand `LISTENING` versetzt, falls der Port, den er verwenden will, noch frei ist. Standardport ist 8, per Aufruf von `setPort()` kann dieser Port aber verändert werden. Erhält der Webserver eine Anfrage, so sendet er als Antwort den String "`Webseite`" an den Sender der Anfrage zurück. Mit `stop()` wird die Ausführung des Webservers beendet, er gibt den belegten Port frei und erreicht den Zustand `STOPPED`.

Dieses Programm wurde zum Testen des SYN-Angriffs und des Serververhaltens bei TCP-Verbindungen (Connect, Dienstausführung) geschrieben. Daher hat der Webserver keine weiterentwickelte Funktion, außer auf Anfragen zu reagieren und einen Port zu belegen, den man connecten kann.

8.4 Implementierung der DoS-Attacken

Wie aus der vorangegangenen Beschreibung der Netzwerk-Implementierung ersichtlich ist, stecken die zur Simulation einer DoS-Attacke verwendeten Schwachstellen in der Implementierung des Servers. Dies ist damit zu erklären, dass wir bisher nur Flooding-Attacken (Smurf-Angriff) und Protokollprobleme (SYN-Attacke) dargestellt haben.

Man kann aber auch eine Programm-Klasse schreiben, bei der die Anwendung eines Exploits zu DoS-Auswirkungen führt. Sie könnte den Server also beispielsweise zum Absturz bringen.

Eine genauere Beschreibung, wie eine solche Programmklasse aussehen müsste, erfolgt hier nicht. Dies ist in dieser Arbeit nicht untersucht worden. Was wir zu diesem Thema hier aber erwähnen möchten ist, dass bei dieser Art von DoS-Attacke das Implementierungsproblem nicht auf Seiten der Erzeugung des Serverausfalls liegt. Durch die Anwendung des Exploits den Server in den Zustand `OFFLINE` zu bringen, wäre spätestens durch Anwendung der Serverfunktion `shutDown()` ohne Probleme möglich. Wie simuliert man dann aber nach einem solchen Servercrash, dass der Server wieder gebootet wird?

Es gibt zur Lösung dieses Problems verschiedene Ansätze: Der Server kann nach einer zufälligen Zeit von sich aus wieder booten, der Bootvorgang kann aber auch durch ein Anwendungsprogramm oder die `Net`-Klasse initiiert werden. Alle genannten Möglichkeiten besitzen Vor- und Nachteile, sie zu untersuchen ist aber nicht Teil dieser Arbeit.

Kommen wir deshalb wieder zu den hier verwendeten Angriffen zurück. Im Abschnitt über die Serverimplementierung wurden die nötigen Grundlagen beschrieben, die wir brauchen, um einen DoS-Angriff zu simulieren. Für den SYN-Angriff

ist dieses der Zähler, der die noch nicht zustandegekommenen Verbindungsaufbauversuche zählt, in Verbindung mit einer maximalen Anzahl solcher Versuche.

Kommt ein SYN-Angriffspaket, so wird der Zähler für Aufbauversuche erhöht. Ist er auf dem definierten Maximalwert, so werden gar keine TCP-Pakete mehr angenommen, bei denen das SYN-Bit gesetzt ist, da keine neuen Verbindungen mehr aufgebaut werden können. Pakete ohne SYN-Bit, aber mit ACK kommen noch an. Ihre Verbindung besteht ja schon, es sind Antworten.

Das Verhaltensdiagramm eines Servers bei einer SYN-Attacke ist in Abb. 43 dargestellt. Dass diese Implementierung gut zu dem von uns entworfenen Modell einer SYN-Attacke passt, beweist die große Ähnlichkeit dieser Abbildung mit der des Modells (Abb. 9).

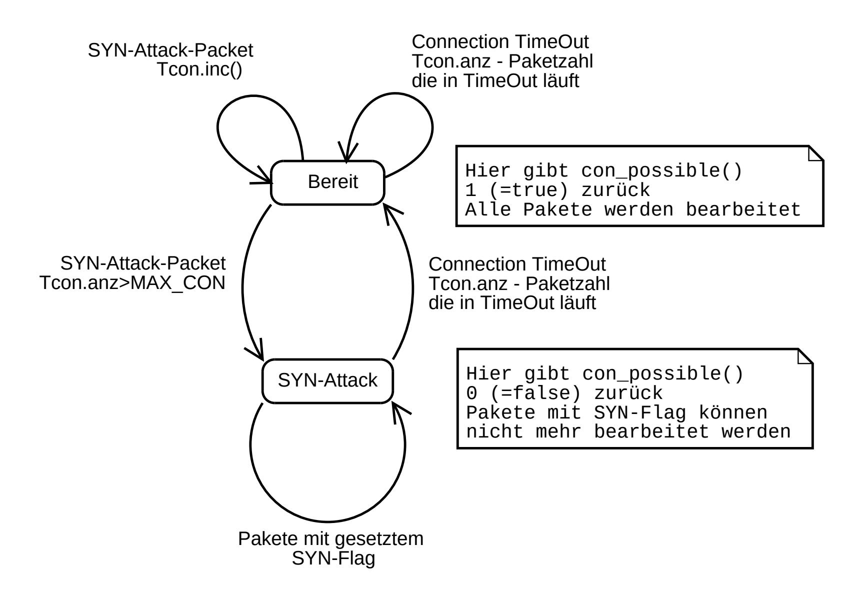


Abbildung 43: Verhalten des Servers bei einer SYN-Attacke

In Abb. 44 ist der Ablauf dargestellt, der beim Empfang eines SYN-Angriffspakets stattfindet. Wenn das Paket beim Server ankommt, wird überprüft, ob die Verbindung angenommen werden kann. In (a) ist dies der Fall, das Paket sorgt also dafür, dass der Aufbauversuch-Zähler erhöht wird. In (b) ist dieser Zähler schon auf dem Maximum. Daher wird dieses Angriffspaket nicht mehr bearbeitet. Es fällt auch selbst dem Angriff zum Opfer.

Zum Testen der SYN-Attacke wurden die Programmklassen `SynAttackTool` und `Webserver` geschrieben, die in 8.3.4 und 8.3.5 beschrieben wurden. Der Angreifer benutzt für seinen Angriff das `SynAttackTool`, das nur auf einem belegten Port eine erfolgreiche Attacke durchführen kann. Auf dem Opfer wurde deshalb ein `Webserver`-Objekt registriert und damit ein Port belegt.

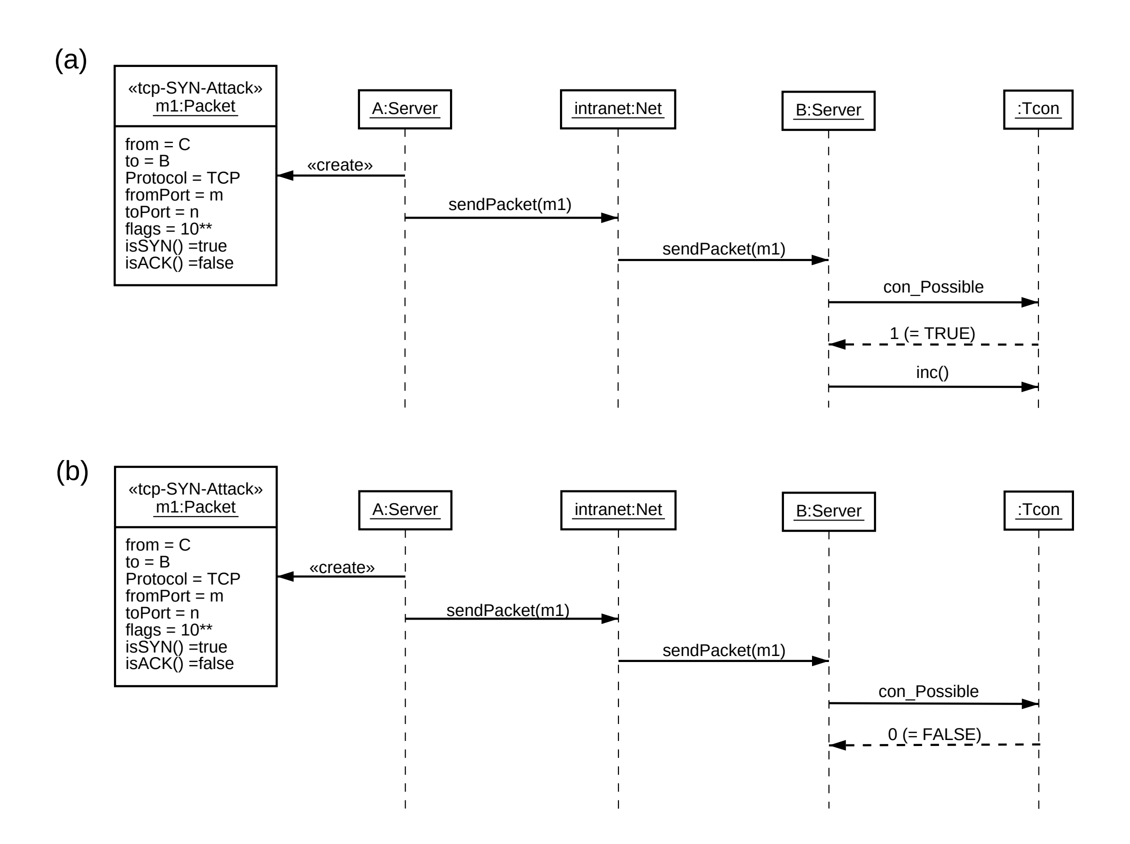


Abbildung 44: Zeitlicher Ablauf einer SYN-Attacke

Bei der Smurf-Attacke ist das Verhalten sehr ähnlich. Zur Veranschaulichung des Serververhaltens verweisen wir hier auf die Darstellung des Modells der Smurf-Attacke (Abb. 10). Die Maximalzahl der bearbeitbaren Pakete wird hierbei vom Server festgelegt, wird sie überschritten, wird kein weiteres Paket mehr bearbeitet, bis die Simulation zum nächsten Zeittakt weitergeht. Um zu simulieren, dass die überzähligen Pakete nicht beim Server ankommen, wird nach Erreichen der maximalen Paketzahl der Rest der Paket-Queue weggeworfen (Listenoperation `clear()`). Der Server wird dann gleichzeitig in den Zustand `OUTOFORDER` versetzt. Beim Zeitwechsel wird der Zähler wieder auf 0 gesetzt. Der Server wird in den Zustand `ONLINE` versetzt und kann wieder Pakete annehmen.

Um diesen Angriff durchzuführen wurde das Tool `Smurf` implementiert. Dieses dient dem Angreifer zur Durchführung einer Smurf-Attacke. Das Opfer braucht keine Programme installiert zu haben, damit der Angriff gelingen kann.

8.5 Gegenmaßnahmen

Der Prototyp bietet drei Gegenmaßnahmen an: Die Konfiguration einer Firewall mit Paketfilter, das Simulieren eines read-only Filesystems auf den Servern und die Installation eines einfachen Intrusion-Detection-Tools (IDT).

Die Implementierung des Paketfilters ist schon im Abschnitt 8.2.4 über den Router erklärt worden. Diese Firewall ist noch sehr einfach, aber man kann mit ihr schon offensichtlich gespoofte Pakete aus dem Netzverkehr herausfiltern (siehe z.B. Regelsatz des Beispiellaufs in 9.3).

Das read-only Filesystem wird mit Hilfe eines Flags in der Klasse `Server` implementiert. Dieses Flag muss vor jedem Installieren eines Programms überprüft werden. Gibt dieses Flag an, dass der Server ein nicht-schreibbares Filesystem besitzt, schlägt die Installation fehl. Sonst funktioniert sie wie in der Klasse `Server` beschrieben.

Das IDT (siehe 8.3.1) ist eine von `Program` abgeleitete Klasse. Beim ersten Start des IDTs wird eine Liste der installierten Programme erstellt. Diese wird zu jedem Zeitschritt mit der aktuellen Liste der Programme verglichen, bei Abweichungen wird eine Neuinstallation des Servers ausgelöst. Um die Liste nach einer gewollten Installation auf den aktuellen Stand zu bringen, ist eine Update-Funktion eingebaut (Aufruf von der `exec()`-Funktion mit speziellem Parameter), die allerdings im Zweifelsfall auch dem Einbrecher in das System zur Verfügung steht. Dieser muss zur Verhinderung seines Entdecktwerdens aber wissen, dass ein IDT installiert ist und wie die Datenbank des IDTs zu aktualisieren ist.

8.6 Viewer

Normalerweise ist die Visualisierungskomponente einer verteilten Simulation eine eigenständige Komponente. Dies geschieht aus Effizienzgründen. Die Rechenzeit, die der Viewer zur Darstellung der Ereignisse braucht, soll den Ablauf der Simulation nicht aufhalten. Deshalb trennt man die passive Darstellungskomponente von den aktiven Simulationsteilen.

In diesem Prototyp ist der Viewer dennoch Teil einer aktiven Simulationskomponente. Hierfür gibt es mehrere Gründe:

1. Da es mehrere an der Simulation beteiligte Angreifer geben kann, ist es einfacher, jedem Angreifer seine Visualisierungskomponente gleich mitzugeben. Jeder Viewer kümmert sich so um „seinen” Angriff, von anderen Netzaktivitäten braucht er nichts zu wissen. Er braucht weder zu wissen, wie viele

Angriffe gleichzeitig erfolgen, noch welche Aktion er welchem Angriff zuordnen muss. Er muss nur entscheiden können, ob eine Aktion zu „seinem" Angriff gehört oder nicht.

2. Um entscheiden zu können, ob eine Aktion zum Angriff seines Angreifers gehört, muss der Viewer eine Art von Erkennungszeichen mit seinem Angreifer ausmachen, mit dem bestimmte Aktionen gekennzeichnet werden. Würde man eigenständige Viewer einsetzen, aber nicht einen, sondern so viele wie Angreifer vorhanden sind, so wären die Aushandlung der Zusammengehörigkeit zwischen Viewer und Angreifer und die Aushandlung des Erkennungszeichens schwierig. Hierfür wäre ein extra Federate notwendig, der eine Managementfunktion zur Kontrolle dieser Zuordnungen ausübt, um sicher zu stellen, dass jeder Viewer genau einem Angreifer zugeordnet wird.

3. Es stellt sich die Frage, ob eine Simulation, die dem Viewer zeitlich gesehen vorauseilt, im Fall einer DoS-Attacke überhaupt sinnvoll ist. Denn bei dieser Simulation ist nicht das Ergebnis des Simulationslaufes das Wichtige, sondern der Weg, wie dieses Ziel erreicht wird.

Deshalb ist der Viewer Teil des Angreifer-Federates. So kann man den Status des Angriffs dieses Angreifers genau verfolgen. Dabei ist der Viewer eine Klasse, die vom Angreifer-Federate benutzt wird. Der Viewer zeigt einen Angreifer-Status, der die momentanen Aktionen des Angreifers darstellt, und einen Netzwerk-Status, in dem der Zustand des Angriffsnetzes und des gewünschten Opfers überwacht werden können. Dazu müssen allgemeine Zustände der beteiligten Server dargestellt werden (online, offline, Überlastung, ...) und Aktionen wie Neuinstallationen. Daneben gibt es die für einen Viewer speziell gedachten Meldungen, wie z.B. die, dass sein Angreifer gerade ein Tool installiert hat oder dass der Server gerade Angriffspakete verschickt, etc. Hierbei sollten auch die Auswirkungen auf das Opfer erkennbar sein. Mehr Feedback als ein „Opfer ist ausgefallen" , „Opfer ist noch voll arbeitsfähig" o.ä. wird von diesem Prototyp aber nicht dargestellt. Der Vorteil dieses Ansatzes ist, dass man kein spezielles Opfer zu simulieren braucht, ein „normaler Netzwerkrechner" als Opfer kann diese Meldungen auch schon erzeugen.

Will man auch die genaueren Auswirkungen auf ein Opfer darstellen, so sollte ein spezielles Modell eines Opfers simuliert werden. Dieses kann dann entweder einen eigenen Viewer besitzen, der genauere Aussagen über den Zustand des Opfers trifft, oder es kann zusätzliche, detaillierte Angaben über seinen Zustand an die RTI schicken, sodass irgendein Viewer, der diese Aussagen interpretieren kann, sie anzeigt.

Für eine Gesamtperspektive über das Netzgeschehen wird aber doch ein von allen anderen Federates unabhängiger Viewer empfohlen. Dieser kann dann auch

als unabhängiger Federate auf einem anderen Rechner als die übrigen Federates zu Präsentationszwecken benutzt werden.

8.7 Angreifer

Für den Angreifer gibt es generell zwei verschiedene Ansätze: Einerseits kann man den Angreifer über das Userkonzept auf jedem beliebigen Server der bestehenden Netzwerke agieren lassen, andererseits kann man ihn als Erbe eines Servers mit einem Hauptprogramm versehen, welches als Arbeit eben einen Angriff durchführt. Man kann die Aktionen aber auch gezielt vom Anwendungsprogramm aus starten.

Für den **Smurf-Angreifer** wurde letztere Möglichkeit gewählt.

Bevor wir nun die Implementierung des Smurf-Angreifers beschreiben, sei darauf hingewiesen, dass dieser Angreifer für ein spezielles Angriffsszenario konzipiert wurde. Dieses wird später in der Beispielsimulation verwendet.

Wie schon in 8.2.7 erwähnt, muss ein Federate sich zuerst bei einer Federation Execution anmelden. Nachdem der Angreifer-Federate dies getan hat, setzt er sein Zeitverhalten wie alle anderen Netzwerke auch auf zeitbestimmend und zeitabhängig.

Während die Federates, die keine Angreifer sind, dann ihr Interesse an Interaktionen wie AngreiferResign, AngreiferJoin oder SimulationEnd bekunden, wird der Angreifer diese höchstens absenden. Da der hier beschriebene Angreifer als alleiniger Angreifer konzipiert wurde, meldet er nur die Veröffentlichung von SimulationEnd bei der RTI an.

Ebenso wie die anderen Netzwerke besteht der Angreifer aus einem kompletten Netzwerk, also je einem Objekt `Net` und `Router`. Da wir einen Angriff in ein anderes Netzwerk planen, braucht das Heimatnetz des Angreifers nur ein `Server`-Objekt zu verwenden. Dieses ist der Basisrechner, von dem der Angreifer aus agiert.

Wir gehen hier davon aus, dass der Angreifer sein Netzwerk kontrollieren kann. Daher lassen wir seinen Router sämtlichen Netzverkehr in beide Richtungen akzeptieren. Die Pakete nach innen interessieren uns momentan nicht, und nach außen würde der Angreifer sich nicht selbst behindern wollen. Als Routerkonfiguration ergibt sich daher `ACCEPT` als Default-Policy bei einem leeren Regelsatz.

Um die Konfiguration des Servers des Angreifers möglichst einfach zu halten, geben wir ihm nur die für den Angriff nötigen Programme: Das Tool, mit dem er seinen Angriff durchführt (ein Objekt vom Typ `Smurf`), und ein Tool namens „Ping", mit dem wir die Netzwerkverbindung zwischen dem Angreifer und Servern anderer Netzwerke testen können.

Im Gegensatz zu den Intranet-Federates hat der Angreifer keine zyklische Arbeit zu erledigen. Er beginnt mit der Installation des Ping- und des Smurf-Tools.

Wenn er nun in die nächste Zeit voranschreitet, so schickt er mit dem Ping-Befehl zu allererst ein Ping Request-Paket an das Opfer und an den Rechner des Amplifier-Netzes mit der HostID 1.

Dann beginnt er seinen Angriff durch Starten des Smurf-Befehls als Service mit den Parametern für das Opfer und das Netz, welches gepingt werden soll. Diesen Befehl lässt der Angreifer nun für 5 Zeitschritte laufen. Dabei werden pro Zeitschritt 10 Ping-Pakete mit der Absendeadresse des Opfers an die Broadcast-Adresse des Amplifier-Netzes gesendet.

Nach diesen 5 Zeiteinheiten stoppt der Angreifer den Angriff durch Stoppen des Smurf-Tools. Dann wartet er noch einmal 5 Zeiteinheiten ab, bis er der Simulation durch Senden der SimulationEnd-Interaktion das Ende signalisiert und dann selbst seine Ausführung beendet. Auch er meldet sich hierzu von der Federation-Execution ab und versucht dann, die Execution zu beenden.

Der Angreifer wartet deshalb noch eine Zeit lang, bevor er das Simulationsende propagiert, damit man in der Simulation auch noch sehen kann, ob sich das Opfer nach Beendigung des Angriffs wieder erholt.

8.8 Opfer

Ein spezielles Opfer wird nicht implementiert. Wie schon in der Konzeption erwähnt, wird hier ein normales Subnetz die Funktion des Opfers übernehmen. Dazu wurde jedem Server die Funktionalität gegeben, seinen Zustand (`ONLINE`, `OFFLINE` bzw `OUTOFORDER`) an die RTI zu senden.

Dadurch, dass jeder Server als Opfer agieren kann, haben die Szenarien den Vorteil, dass auch Reaktionen auf den Angriff, wie z.B. ein Gegenangriff auf den vermeintlichen Angreifer, und ihre Auswirkungen auf die Erreichbarkeit des Servers darstellbar sind, ohne dazu noch zusätzliche Opfer ausweisen zu müssen. Der Angreifer kann also beliebige falsche Adressen generieren. Wenn die Absendeadresse zu einem simulierten Server gehört, so kann dieser sich bei Reaktionen des Opfers entsprechend verhalten.

8.9 Helfer

Die Helfer sind einfach vom Angreifer bestimmte Server aus den Teilnetzen. Da man nicht weiß, welche der Server ausgesucht werden, müssen also alle Server die Funktionen beherrschen, die ein Helfer braucht, um an einen Viewer seine Statusinformationen weiterzugeben, anhand derer der Viewer den Zustand des Servers darstellen kann. Dabei erscheint es nicht sinnvoll, die Zustände „Clean", „Master",etc., wie sie in 6.4 definiert sind, im Server direkt zu codieren, da bei der Simulation mehrerer Angriffe gleichzeitig so Verwirrung oder gar Fehler auftreten können. Viel eher sollte der Server bestimmte Vorgänge wie z.B. „Installation eines Angriffstools" mit einem bestimmten Code des zugehörigen Angreifers als RTI-Interaktion an den Viewer schicken, der diese dann auswertet.

Aus diesem Grund senden die Server Statusmeldungen, wenn sie neu installiert werden. Angriffstools sollten, wenn sie gestartet oder gestoppt werden, jeweils eine Meldung „Tool aktiviert" bzw „Tool gestoppt" mit dem Code ihres Angreifers und ihrer eigenen Funktion (Master/Daemon) an die RTI senden.

8.10 Schwachpunkte des Prototyps

Bei der Implementierung des Prototyps wurden verschiedene Problembereiche erkannt, die für die grundsätzliche Funktion nicht relevant waren, und deshalb im Rahmen dieser Arbeit nicht näher untersucht wurden. Sie sollen hier jedoch kurz dargestellt werden, wobei auch Lösungsvorschläge angegeben werden.

8.10.1 Reihenfolge der Pakete abhängig von HostID der Sender

Pakete kommen nach Servern sortiert ins Netz, daher erfolgt eine Benachteiligung der Server mit höherer HostID. Dadurch ändern sich zwar nicht die Auswirkungen auf den Server, er wird bei einer DoS-Attacke auch so in die Knie gezwungen, aber mögliche Sichten von außen auf das Opfer werden hierdurch beeinflusst.

Beispiel: Der Angreifer hat eine höhere HostID als der User, der Anfragen an den Webserver stellt: Die Pakete des Users werden immer dann beantwortet, wenn der Webserver wieder neue Pakete empfangen kann, weil einige Verbindungen durch TimeOut freigeworden sind. Erst nach der Bearbeitung der http-Anfragen werden die Angriffspakete bearbeitet, die wieder weitere Verbindungswünsche verhindern.

Hat der Angreifer die niedrigere HostID, so werden zuerst die eben wieder freigewordenen Verbindungen belegt, die Anfragen des Users an den Webserver

fallen dem Angriff zum Opfer.

Mögliche Lösungsansätze:

1. Die Pakete werden durchmischt. Man könnte sie z.B. im Net-Objekt sammeln. Nachdem alle Server aktiv waren (`doWork()` wurde ausgeführt, die hierbei erzeugten Pakete werden im Net-Objekt des Netzwerks gesammelt), kann eine Durchmischung der Pakete erfolgen, die zu einer zufälligen Reihenfolge führt, dann erst werden Pakete an den Bestimmungsort (Router bzw. Server) weitergereicht.

2. Das Intranet wird auch als parallele Simulation aufgebaut, bei der die Reihenfolge der Pakete dann also von der Simulation bestimmt wird.

8.10.2 Teilnetze mit gleicher NetzID möglich

Der bestehende Prototyp bietet keine Möglichkeit, die Erzeugung mehrerer Teilnetze mit der selben NetzID zu verhindern. An sich wäre es vielleicht aus Gründen vergleichender Simulation sogar wünschenswert, zwei Netze mit gleicher NetzID zu haben, aber für diesen Prototyp sehen wir es als Konfigurationsfehler an.

Der Grund dafür ist, dass der Angreifer-Viewer des Prototyps zwei Server mit gleicher IP nicht unterscheiden kann, da er von einer eindeutigen Benennung der erreichbaren Rechner ausgeht. Daher würde er die von den gleichbenannten Servern erhaltenen Statusmeldungen, die lediglich nach IP-Adressen verwaltet werden, miteinander vermischen. Der Unterschied, der durch zwei gleichbenannte Netze simuliert werden sollte, wäre also mit dem bestehenden Viewer des Angreifers nicht darstellbar.

Lösungen dieses Problems können in zwei Richtungen gehen: Entweder wird eine Möglichkeit entwickelt, die verhindert, Teilnetze mit gleicher ID zu erzeugen, oder man entwickelt einen Viewer, der durch gleiche IPs nicht in Unordnung gerät.

Mögliche Lösungsansätze:

1. Will man gleiche NetzIDs verhindern, würde es sich anbieten, einen Federate zu verwenden, der Verwaltungsaufgaben erledigt. Zu diesen könnte dann auch die Verwaltung von NetzIDs gehören. Diese Verwaltung könnte folgendermaßen aussehen: Wird ein Intranet-Federate erzeugt, so kann er bei dem Management-Federate nach einer freien NetzID fragen, die er dann verwendet. Oder er kann einen Wunsch nach einer bestimmten NetzID äußern. Ist diese frei, so bekommt er sie, ansonsten kann er eine andere ID zugeordnet bekommen oder seine Ausführung abbrechen.

2. Will man gleiche NetzIDs bewusst zulassen, so müsste man dem Viewer eine andere Möglichkeit geben, die erhaltenen Meldungen zu ordnen. Jedes Netzwerk bräuchte eine eindeutige Identifizierung neben der (eben nicht eindeutigen) NetzID.

8.10.3 Keine Festlegung von maximaler Bandbreite

Jeder Server hat eine festgelegte maximale Anzahl von Paketen, die er pro Zeitabschnitt empfangen und auch bearbeiten kann. Senden kann ein Server in diesem Zeitabschnitt bisher aber eine beliebige Anzahl von Paketen. Dieses Verhalten ist nicht realistisch. Bei der Implementierung der Programme und des Angreifers muss daher darauf geachtet werden, sinnvolles Verhalten zu konfigurieren.

Zu wünschen wäre deshalb eine Beschränkung der Sendebandbreite, bei der der Server eine bestimmte Obergrenze bei der Paketgenerierung nicht überschreiten kann. Diese Grenze wäre im sinnvollen Verhältnis zu der Empfangsbandbreite festzulegen.

Da es dabei nicht reicht, die Anzahl der Pakete, die versendet werden können, zu beschränken, sollte noch eine Größenangabe für Pakete eingeführt werden, die dann bei der Feststellung, welche und wieviele Pakete verschickt werden können, zu berücksichtigen wäre.

Mögliche Lösungsansätze:

1. Ein Server muss selbst kontrollieren, ob er noch Pakete verschicken darf. Dabei tritt aber das Problem auf, dass auf dem Prototyp-Server die Programme sequentiell ausgeführt werden und nicht parallel. Somit wäre ein dem Problem mit der ankommenden Paketreihenfolge in 8.10.1 ähnelndes neues Problem entstanden.

2. Das Netz könnte die Paketrate überwachen. Damit müsste das Netz die Pakete des Servers sammeln. Wäre die Paketmenge zu groß, müssten nach noch zu bestimmenden Regeln die überschüssigen Pakete verworfen werden. Erst dann dürfte das Netz die Pakete weiterleiten.

Es wäre ebenso anzuraten, eine maximale Durchgangsrate für Router festzulegen, damit auch hier eine weitere Annäherung des Prototyps an die Realität erreicht werden kann.

9 Beispiellauf einer DDoS-Simulation

Nachdem wir ausführlich die Theorie erläutert haben, die unserem Prototyp zugrunde liegt, wollen wir nun zeigen, wie er in der Praxis arbeitet.

Wir wählen dazu als Angriff eine Smurf-Attacke.

Weiter wollen wir an diesem Beispiel die Wirksamkeit einer Firewall zur Verhinderung von Smurf zeigen. Dazu werden wir zunächst ein Szenario erstellen, das ohne Gegenmaßnahmen einen solchen Angriff erfolgreich zulässt (9.1). Die Beschreibung der Simulation erfolgt dazu in Abschnitt 9.2. In Abschnitt 9.3 werden wir eine leicht abgewandelte Simulation durchführen, bei der wir mittels einer Firewallregel verhindern, dass der Angreifer mit seiner Attacke erfolgreich ist. Dabei werden wir die nötigen Firewallregeln darstellen und auf die Unterschiede zum ersten Simulationslauf eingehen.

9.1 Szenario

Für die Simulation des Smurf-Angriffs benötigen wir 3 Federates: den Smurf-Angreifer und zwei Intranet-Federates, eines als Opfernetz, das andere als Amplifier-Netz, welches den vom Angreifer erzeugten Traffic verstärkt.

Wir beschreiben nun die Konfiguration des entworfenen Szenarios.

9.1.1 Amplifier-Netz

Das Amplifier-Netz wird mit der NetzID 1 initialisiert. Da zu einem erfolgreichen Smurf-Angriff das Opfer mindestens 100 Pakete erhalten muss und pro Zeiteinheit der Angreifer 10 mal die Broadcastadresse dieses Amplifier-Netzes anpingt, brauchen wir mindestens 10 Rechner, die auf diesen Ping antworten. Wir konfigurieren unser Amplifier-Netz deshalb auf 10 Rechner.

Als Konfiguration für den Router geben wir für diesen Simulationslauf keine Firewallregel mit, der Router wird deshalb jeglichen Netzwerktraffic weiterleiten, sowohl in das Netz hinein, als auch hinaus.

Auf den Rechnern dieses Netzwerks ist kein Programm installiert, da das zur Simulation eines Smurf-Angriffs nicht nötig ist und wir aus Gründen der Übersichtlichkeit auf nicht Benötigtes verzichten.

9.1.2 Opfernetz

Das Opfernetz erhält die NetzID 2. Die Anzahl der Server dieses Netzes hat auf den Erfolg der Attacke keine Auswirkungen. Daher werden wir das Netz möglichst klein gestalten, dann bleibt es übersichtlicher für die Auswertung. Um zu zeigen, dass der Angriff sich nur auf den angegriffenen Opferrechner auswirkt, werden wir neben dem Opfer aber doch einen zweiten Rechner in diesem Netz betreiben.

Auch auf diesen beiden Rechnern installieren wir aus dem selben Grund wie beim Amplifier-Netz keine Programme.

9.1.3 Angreifer

Das Netz, in dem der Angreifer arbeitet, erhält die NetzID 3. Des weiteren bleibt die Konfiguration des Angreifernetzes wie schon in 8.7 beschrieben: keine Firewallregeln auf dem Router und nur ein Rechner im Netz.

9.2 Erfolgreicher Smurf-Angriff

Um die Simulations-Traces zu verstehen, sollten wir zunächst einige Darstellungspunkte klären. Sonst verliert man leicht den Überblick, welche Aktion zu welcher Zeit stattfindet, wann genau der Zeitwechsel stattgefunden hat oder welche Meldung von welchem Server erzeugt wurde.

Deshalb zum besseren Verständnis der folgenden Trace-Auszüge:

- **Zeitwechsel**

  Jeder Zeitwechsel wird durch Ausgabe der Zeile

  ```
  ***** New Time:  x.0000000000 *****
  ```

  angezeigt, wobei `x` durch die entsprechende aktuelle Simulationszeit ersetzt wird. Alle Ausgaben vor dieser Zeile stammen aus der Zeit vor dem Zeitwechsel, alle darauf folgenden von danach.

- **Betroffener Server**

  Die Ausgaben, die von den Servern erzeugt werden, werden durch

  ```
  -- IP --
  ```

mit `IP` als der IP des jeweiligen Servers eingeleitet. Für den Router erscheint statt einer IP der String `Router`. Nachdem der letzte Server des Netzes seine Arbeit erledigt hat, erscheint die Meldung

```
-- net finished --
```

Zu den Meldungen gehört, dass jeder Server und auch der Router angibt, wieviele Pakete er aktuell zu bearbeiten hat. Dies wird mit `packets to handle:` gefolgt von der Paketzahl gemeldet.

Dieses sind nur die Meldungen, die der Server während der Ausführung von `doWork()` macht, also während der nicht interaktive Teil seines Programms abgearbeitet wird. Alle Meldungen, die danach generiert werden, also durch Befehlsausführung aus dem Hauptprogramm heraus, folgen erst nach der „`-- net finished --`" Meldung.

Ein weiterer Hinweis zu dieser Simulation: Wir starten hier die Federates in der Reihenfolge Amplifier-Netz, Opfernetz und dann den Angreifer.

9.2.1 Angreifer

Der Simulationslauf beginnt mit den Meldungen, dass der Angreifer sich an der Simulation angemeldet hat. Da der Angreifer als Letzter der Federation Execution beitritt, existiert diese schon, wenn er versucht, sie zu erstellen. In dieser Simulation erhält der Angreifer zusammen mit den Bestätigungen über das Erreichen seiner gewünschten Zeitverhalten (timeRegulation und timeConstrained) die initiale Simulationszeit 6 gemeldet. Der Angreifer erhält von der RTI eine FederateID 5, die nicht mit der NetzID (3) verwechselt werden darf.

```
ANGREIFER: Creating Execution... already existing! Going on...
ANGREIFER: Joining Execution...  joined. My ID: 5
ANGREIFER: timeRegulationEnabled. New Time: 6.0000000000
ANGREIFER: timeConstrainedEnabled. New Time: 6.0000000000
```

Nach dem Startvorgang meldet sich nun der Angreifer mit seiner NetzID und bestätigt, dass er einen Rechner zum Arbeiten gefunden hat. Es folgt die Ausgabe der bestehenden Firewallregeln des Angreifernetzes, welche in diesem Fall mit Ausgabe der Default-Policy `ACCEPT` abschließen.

```
 Attacker-Simulation for Subnet #3
------------------------------------

OK, i got an operating Basis: 3.1

Firewall Ruleset:
ACCEPT
```

Nun beginnt der Server mit der Arbeit. Im Folgenden ist zu sehen, dass der Router in diesem Zeitabschnitt keine Pakete zu bearbeiten hat. Ebenso hat auch der Rechner des Angreifers (`3.1`) keine Pakete erhalten. Zuerst führt der Angreifer einen Ping auf die zwei Rechner `1.1` und `2.1` aus. Dieser Ping ist bei dieser Simulation noch nicht so wichtig, aber er wird bei einer Simulation mit Firewallregeln zeigen, dass noch Pakete die Firewall passieren können. Dieser Ping wird nur dieses eine Mal ausgeführt. Danach wird der Befehl Smurf ausgeführt, der eine Meldung für jedes abgeschickte Paket ausgibt. Wir sehen so 10 Pingpakete, die an die Adresse `1.255` (Netz 1, Broadcastadresse) geschickt werden und dabei `2.1` als Absender angegeben haben.

```
 ***** New Time: 7.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 3.1 ---
3.1 packets to handle: 0

 --- Net finished ---
Befehl ping wird ausgeführt auf 3.1
ping 1.1
ping 2.1
Befehl smurf wird ausgeführt auf 3.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
```

Im nächsten Zeittakt sehen wir dann, dass der Router die 2 Ping- und die 10 Angriffspakete erhalten hat. Der Angreiferrechner sendet weiter noch die Angriffspakete, jetzt aber zeitlich bei der Ausführung von `doWork()`.

```
 ***** New Time: 8.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 12

 --- 3.1 ---
3.1 packets to handle: 0
Befehl smurf wird ausgeführt auf 3.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1

 --- Net finished ---
```

Zu Simulationszeit 9 bis 11 passieren nur noch die Angriffspakete den Router. Bei Zeittakt 12 erhält der Router dann wieder zwei Pakete mehr: Neben den Angriffspaketen treffen die Antworten auf die Ping-Anfragen ein. Diese werden zum Angreifer weitergeleitet und treffen zu Zeitpunkt 12 dort ein.

```
 ***** New Time: 12.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 12

 --- 3.1 ---
3.1 packets to handle: 0
Befehl smurf wird ausgeführt auf 3.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
```

```
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1
ping to 1.255, Spoofed IP:2.1

 --- Net finished ---
Attack finished!
```

Bei Zeitpunkt 12 sendet der Angreifer das letzte Mal Angriffspakete. Zu Takt 13 zeigt er keine Aktivitäten mehr. Der Router muss dennoch die Pakete, die der Angreifer den Takt davor abgeschickt hat, bearbeiten. Die zwei Pakete, die Rechner `3.1` erhält, sind die Ping-Replys.

```
 ***** New Time: 13.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 10

 --- 3.1 ---
3.1 packets to handle: 2

 --- Net finished ---
```

Noch einen Takt später hat auch der Router keine Pakete mehr zu bearbeiten.

```
 ***** New Time: 14.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 3.1 ---
3.1 packets to handle: 0

 --- Net finished ---
```

Das Verhalten des Angreifers ändert sich nicht mehr, bis er zu Simulationszeit 22 die Ausführung der Simulation als beendet deklariert und dies den anderen Federates mitteilt. Danach beendet er sich.

```
 ***** New Time: 22.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 3.1 ---
3.1 packets to handle: 0

 --- Net finished ---
Attack finished!
Telling others I've finished.
ANGREIFER: Resigning Execution...  resigned.
ANGREIFER: Destroying Execution... Execution already destroyed.
```

9.2.2 Amplifier-Netz

Beschreiben wir nun, welches Verhalten das Amplifier-Netz während unserer Simulation zeigt. Auch dieser Federate beginnt mit den Meldungen, die schon vom Angreifer her bekannt sind. Allerdings beginnt er bei Simulationszeit 0, da er als erstes der Execution beigetreten ist.

```
INTRANET: Creating Execution...    created.
INTRANET: Joining Execution...     joined. My ID: 1
INTRANET: timeRegulationEnabled. New Time: 0.0000000000
INTRANET: timeConstrainedEnabled. New Time: 0.0000000000
```

Nun können wir die Netzinitialisierung verfolgen. Wir sehen dabei, wie Server `1.1` bis `1.10` angelegt und in das Netz eingefügt werden.

```
 Intranet-Simulation for Subnet #1
------------------------------------

Added server 1.1 to Intranet.
Added server 1.2 to Intranet.
Added server 1.3 to Intranet.
Added server 1.4 to Intranet.
Added server 1.5 to Intranet.
Added server 1.6 to Intranet.
Added server 1.7 to Intranet.
Added server 1.8 to Intranet.
Added server 1.9 to Intranet.
Added server 1.10 to Intranet.
```

Danach beginnt die Arbeit des Netzwerkes. In diesem Fall besteht diese Arbeit daraus, auf eingehende Pakete zu warten. Der Output des Traces sieht deshalb von Simulationszeit 0 bis 8 folgendermaßen aus (Server : `1.1` und `1.2` stehen stellvertretend für alle Rechner des Netze):

```
 ***** New Time: 0.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 1.1 ---
1.1 packets to handle: 0

 --- 1.2 ---
1.2 packets to handle: 0
```

Am Ende jedes Zeittaktes gibt das Netz einen Status über seine Rechner aus. Dieser ändert sich im Laufe der Simulation nicht, da aus diesem Netz kein Rechner angegriffen wird.

```
1.1 ONLINE
1.2 ONLINE
1.3 ONLINE
1.4 ONLINE
1.5 ONLINE
1.6 ONLINE
1.7 ONLINE
1.8 ONLINE
1.9 ONLINE
1.10 ONLINE
```

Zu Simulationszeit 9 empfängt der Router dann die vom Angreifer gesendeten Pakete, die Server sind immer noch unbeschäftigt:

```
 ***** New Time: 9.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 11

 --- 1.1 ---
1.1 packets to handle: 0

 --- 1.2 ---
1.2 packets to handle: 0
```

Ab Simulationszeit 10 beginnen dann die Rechner des Netzes, die Broadcast-Pings zu beantworten, Rechner `1.1` muss zu Zeit 10 noch zusätzlich den einmaligen Ping des Angreifers bearbeiten.

```
 ***** New Time: 10.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 10

 --- 1.1 ---
1.1 packets to handle: 11

 --- 1.2 ---
1.2 packets to handle: 10
```

Zu Simulationszeit 11 ist dann ein sehr großes Paketaufkommen beim Router des Netzwerks zu beobachten. Dieser bearbeitet nun nicht nur die Broadcast-Pings des Angreifers, sondern auch alle Ping-Antworten der Rechner seines Netzes.

```
 ***** New Time: 11.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 111

 --- 1.1 ---
1.1 packets to handle: 10

 --- 1.2 ---
1.2 packets to handle: 10
```

Nachdem auch die Antwort auf den einmaligen Ping des Angreifers das Netz wieder verlassen hat, pendelt sich das Paketaufkommen des Routers bei 110 Paketen ein.

```
 ***** New Time: 12.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 110

 --- 1.1 ---
1.1 packets to handle: 10

 --- 1.2 ---
1.2 packets to handle: 10
```

Dieses Aufkommen ändert sich erst zu Simulationszeit 15 wieder. Ab dieser Zeit kommen keine weiteren Pakete mehr vom Angreifer, zu Zeit 15 und 16 werden deshalb nur noch die 100 Pakete aus dem eigenen Netz verschickt, ab Zeit 17 findet keine Kommunikation mehr statt.

Zu Simulationszeit 22 wird das Zeichen zum Simulationsende empfangen, der Federate beendet seine Ausführung.

```
Got end of simulation
INTRANET: Resigning Execution...  resigned.
INTRANET: Destroying Execution... Execution already destroyed.
```

9.2.3 Opfernetz

Nun fehlt uns nur noch, wie der angegriffene Rechner die Simulation erlebt.

Das Opfernetz tritt der Simulation zur Zeit 4 bei und legt dann seine zwei Server an.

```
INTRANET: Creating Execution...    already existing! Going on ...
INTRANET: Joining Execution...     joined. My ID: 3
INTRANET: timeRegulationEnabled. New Time: 4.0000000000
INTRANET: timeConstrainedEnabled. New Time: 4.0000000000


 Intranet-Simulation for Subnet #2
------------------------------------

Server 2.1 angelegt!
Server 2.2 angelegt!
```

Während der folgenden Zeit passiert in diesem Netz nichts. Es gehen keine Pakete ein und alle Server sind online. Dieser Zustand hält bis zu Simulationszeit 8 an.

```
 ***** New Time: 5.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 2.1 ---
2.1 packets to handle: 0
```

```
 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 ONLINE
2.2 ONLINE
```

Zu Simulationszeit 9 trifft ein Ping-Paket von Angreifer ein. Erst passiert es den Router, dann wird es vom Opfer beantwortet, bei Zeit 11 verlässt die Antwort das Netz wieder über den Router:

```
 ***** New Time: 9.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 1

 --- 2.1 ---
2.1 packets to handle: 0

 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 ONLINE
2.2 ONLINE


 ***** New Time: 10.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 2.1 ---
2.1 packets to handle: 1

 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 ONLINE
2.2 ONLINE
```

```
 ***** New Time: 11.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 1

 --- 2.1 ---
2.1 packets to handle: 0

 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 ONLINE
2.2 ONLINE
```

Zu Simulationszeit 12 erhält der Router des Opfernetzes die Angriffspakete. Noch ist bei den Servern im Netz alles ok. Da aber der Router die Ping-Pakete an das Opfer weiterleitet, erfährt dieses zu Zeit 13 eine Überlastung:

```
 ***** New Time: 12.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 100

 --- 2.1 ---
2.1 packets to handle: 0

 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 ONLINE
2.2 ONLINE


 ***** New Time: 13.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 100

 --- 2.1 ---
2.1 packets to handle: 100
```

```
Too many packets arrived! Can’t handle every packet.

 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 OUTOFORDER
2.2 ONLINE
```

Zu Simulationszeit 18 ebbt die Flut der Pakete dann ab. Der Router bekommt keine neuen Pakete mehr, das Opfer muss aber noch welche bearbeiten.

```
 ***** New Time: 18.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 2.1 ---
2.1 packets to handle: 100
Too many packets arrived! Can’t handle every packet.

 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 OUTOFORDER
2.2 ONLINE
```

Zu Simulationszeit 19 ist der Angriff vorüber. Das Opfer hat sich wieder erholt.

```
 ***** New Time: 19.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 0

 --- 2.1 ---
2.1 packets to handle: 0

 --- 2.2 ---
2.2 packets to handle: 0

 --- Net finished ---
2.1 ONLINE
2.2 ONLINE
```

Das Ende der Simulation ist für das Opfernetzwerk identisch mit dem Amplifier-Netzwerk. Der Federate erhält das Zeichen zum Beenden und zieht sich zu Zeit 22 aus der Simulation zurück

```
Got end of simulation
INTRANET: Resigning Execution...     resigned.
INTRANET: Destroying Execution...    destroyed.
```

9.2.4 Der Simulationslauf

Nachdem nun die einzelnen Federates auf ihr Verhalten untersucht worden sind, bringen wir die Simulation in einen Zusammenhang:

Zu Zeit 7 schickt der Angreifer seine ersten Pakete. Diese erreichen seinen Router zu Zeit 8. Eine Zeiteinheit später (Zeit 9) kommen die Pakete beim Router des Amplifiernetzes an und werden zur nächsten Zeit 10 beantwortet. Die Flut der Ping-Replys passiert den Router des Amplifier-Netzes zu Zeit 11 und kommt zu Zeit 12 beim Router des Opfers an. Ab Simulationszeit 13 ist das Opfer für die Dauer des Angriffs außer Gefecht gesetzt.

9.3 Verhinderter Smurf-Angriff

Wir verwenden nun fast die gleiche Konfiguration für die Simulation noch einmal. Die einzige vorgenommene Änderung ist, dass wir dem Router des Amplifier-Netzwerks Firewallregeln geben. Diese verhindern, dass der Router Anfragen auf die Broadcastadresse in das Amplifier-Netz hineinlässt.

Der verwendete Regelsatz sieht folgendermaßen aus:

```
DENY
*.*  1.255  *  *  ALL  DROP
1.*   1.*   *  *  ALL  DROP
1.*   *.*   *  *  ALL  ACCEPT
*.*   1.*   *  *  ALL  ACCEPT
```

Damit sollen alle Pakete, die als Zieladresse die Broadcastadresse des Netzes haben, weggeworfen werden. Ebenso werden die Pakete, die als Absender und Ziel das Netz haben, verworfen werden. Diese müssen eine falsche Absendeadresse haben, da sie sonst nicht über den Router geschickt würden. Der Router lässt auch

keine Pakete mit falscher eigener Netzadresse nach draußen, da entweder die Absendeadresse oder die Zieladresse die eigene NetzID enthalten muss. Alle anderen Pakete dürfen das Netz verlassen oder werden hineingelassen.

Dass die Firewall damit nicht einfach alle Pakete blockt, wird die Simulation zeigen: Das Ping-Paket, mit dem der Angreifer die Verbindung zum Rechner aus dem Amplifier-Netzwerk und zum Opfer testet, schafft immer noch sowohl Hin- als auch Rückweg, wie der Trace zeigen wird.

Auf den Angreifer Trace werden wir nicht noch mal eingehen. Dieser ändert sich im Vergleich zum ersten Simulationslauf nicht.

Das Opfer brauchen wir auch nicht genauer zu untersuchen. Bei Betrachtung des Traces kann man sehen, dass der einzelne Ping des Angreifers wie bei der ersten Simulation ankommt, bearbeitet wird, und die Antwort zurückgesendet wird. Danach passiert gar nichts mehr, da keine Angriffspakete mehr das Opfernetz erreichen. Den Grund hierfür werden wir bei Betrachtung das Amplifier-Netzes sehen.

Wie schon in der Simulation zuvor bearbeitet der Router zum Simulationszeitpunkt 9 die Pakete des Angreifers. Von den 11 erhaltenen Paketen werden 10 verworfen. Da der Angreifer 10 gleiche Angriffspakte und einen korrekten Ping gesendet hat, kann man davon ausgehen, dass nur der Ping, der direkt an Rechner `1.1` gesendet wurde, durch die Firewall kommt. Das entspricht dem Regelsatz der Firewall.

```
 ***** New Time: 9.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 11
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.

 --- 1.1 ---
1.1 packets to handle: 0

 --- 1.2 ---
1.2 packets to handle: 0
```

Wie wir sehen können, geht das Paket, das von der Firewall akzeptiert wurde, tatsächlich nur an Rechner `1.1`:

```
 ***** New Time: 10.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 10
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.

 --- 1.1 ---
1.1 packets to handle: 1

 --- 1.2 ---
1.2 packets to handle: 0
```

Das Antwortpaket auf den korrekten Ping verlässt das Netzwerk unbehindert. Von den 11 Paketen, die der Router bearbeiten muss, werden nur 10 blockiert.

```
 ***** New Time: 11.0000000000 *****

 --- Router ---
ROUTER: packets to handle: 11
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
FILTER: Packet rejected: dropped.
```

```
 --- 1.1 ---
1.1 packets to handle: 0

 --- 1.2 ---
1.2 packets to handle: 0
```

Wir können also sehen, dass alle Pings auf die Broadcastadresse geblockt werden. Daher taugt das Netz nicht mehr als Verstärker für den Angriff. Das Gegenteil ist sogar der Fall, denn alle Pakete, die zur Erzeugung von direkten Angriffspaketen gedacht waren, werden verworfen. Kein einziges Angriffspaket kommt noch beim Opfer an.

10 Ausblick

Bei der Anwendung des Prototyps hat sich gezeigt, dass durch die Simulation die Gefahren von DoS-Attacken und die Schutzmechanismen in ihrer Wirkungsweise gut dargestellt und verfolgt werden können.

Es ergaben sich bei den Testläufen aber noch viele weitere Ideen, wie der Prototyp als Anwendungssystem weiterentwickelt werden könnte (siehe hierzu auch 8.10).

In Kapitel 1 haben wir z.B. als Motivation für diese Arbeit gesagt, dass unter anderem Entscheidungsträger in Firmen mit einer Simulation über Gefahren im Internet aufgeklärt werden könnten. Für dieses Ziel müssen insbesondere die Darstellungsmöglichkeiten erweitert und verfeinert werden. Die Entscheidungsträger müssen erkennen können, dass ihnen ein Abbild ihres eigenen Netzes präsentiert wird, dass die Gefahren auch genau dieses Netz betreffen und dass es wichtige Schutzmechanismen gibt.

Die Präsentation vor Entscheidungsträgern muss eine geeignete Auswertung der Netzaktivität liefern, die anschaulich und übersichtlich ist. Entscheidungsträger wollen keine kryptisch anmutenden Tabellen zu Gesicht bekommen und sich diese dann noch von jemand anderem interpretieren lassen, der unter dem Schlagwort „Steigerung der Sicherheit" sein Produkt vermarkten will.

Auch die Benutzerfreundlichkeit der Simulation kann noch verbessert werden. Es sollte ein Framework entwickelt werden, in dem man auf einfachem Wege verschiedene Szenarien zusammenstellen kann. Dazu würde sich eine Oberfläche anbieten, mit der die Netzwerke verwaltet werden könnten. Erst würde man seine Netze designen, den Angreifer und das Opfer wählen und verschiedene andere mögliche Entscheidungen über die Konfiguration des Szenarios treffen. Dann würde man mit der Option „Simulation starten" die Ausführung der Simulation beginnen.

Gegen eine solche Anwendung spricht aber, dass sie die verteilte Ausführung der Simulation behindern würde. Wie bringt man einfache Konfigurierbarkeit, die Möglichkeit der verteilten Simulation und das Starten der Federates auf einfache Weise unter eine Hut? Wenn man jeden Federate mit einer eigenen Oberfläche für die Konfiguration ausstattet, die manuell bedient werden muss, kann man kaum große Szenarien entwerfen. Jeden Federate mit einer bestimmten Konfiguration zu versehen und so eine Sammlung von vielen statischen, verschiedenen Teilen zu haben wird zu unübersichtlich. Daher erscheint es sinnvoll, Federates mit Konfigurationsfiles aufrufen zu können.

Eine interessante Anwendung wäre ein interaktiver Federate, z.B. ein interaktiver Angreifer oder ein interaktives Netzwerk, bei dem der Benutzer versuchen soll,

den Angriff zu erkennen und zu verhindern. Der Anwender müsste beispielsweise Firewallregeln entwerfen und auf den Router spielen, damit der Angriff abgeblockt wird. So, wie ein Systemadministrator es in der Realität auch zu tun hätte.

So ein Ansatz wirft die Frage auf, ob man eine reine Visualisierung und damit Aufklärung über die Problematik der Gefahren im Internet anstrebt, oder ob man nicht durch diese aktive Teilnahme am Geschehen der Simulation weit mehr erreichen kann. Mit interaktiven Angreifern und Netzwerken könnten auch mehrere Personen gleichzeitig trainiert werden. Der Verteidiger eines Netzes kann dann das Vorgehen des Angreifers nicht vorhersehen.

Aber auch konzeptuell sollten noch weitere Wege beschritten werden. Das schon im Kapitel 7.4 über das Konzept des Servers erwähnte Einbeziehen von simulierten Usern und eine Vergabe von Nutzungsrechten auf Servern scheint weitere Möglichkeiten zu erschließen. Damit könnte man auch geeignete Simulationen der Verbreitung von E-Mail-Viren verwirklichen oder die Auswirkungen eines Einbruchs in einen Rechner genauer darstellen.

11 Zusammenfassung

Die gestellte Aufgabe war es, eine Simulation von Bedrohungen im Internet zu entwickeln und mögliche Auswirkungen anhand eines simulierten Beispielszenarios darzustellen. Als Bedrohung wurden DoS-Angriffe gewählt, deren Auswirkungen zu Serverausfällen und Unerreichbarkeits-Problemen im Internet führen können.

Um die gestellte Aufgabe zu lösen, haben wir uns zuerst mit dem Begriff der DoS-Attacken vertraut gemacht und uns mit einigen Möglichkeiten der Abwehr solcher Angriffe befasst. Wir haben uns weiter mit den Grundlagen der Simulation des Internets beschäftigt und mit den Problemen, die eine Internetsimulation aufwirft.

Für die Programmierung des Simulations-Prototyps steht eine Laufzeitumgebung gemäß der HLA-Spezifikation zur Verfügung. Wir haben einen Überblick über die Funktionalitäten gegeben, die wir bei der Verwirklichung unserer verteilten Simulation einsetzen können. Wir haben ihre Möglichkeiten beschrieben und auch ihre praktische Verwendung erläutert.

Mit den so gewonnenen Erkenntnissen haben wir Modelle erstellt, die ein Abbild realer Netzwerkkomponenten und DoS-Abläufe darstellen. Das Verhalten der Implementierung des Prototyps muss sich dann an diesen Modellen ausrichten. Zur Darstellung der Modelle haben wir UML-Diagramme verwendet.

Wir haben eine Konzeption für die Simulation erstellt, die als Basis für die Implementierung des Prototyps in C++ dient. Wir haben ein modulares System konzipiert, damit Variabilität und Erweiterbarkeit des entstehenden Prototyps nicht behindert werden. Weiter sind wir auf konzeptuelle Fehlschläge eingegangen und haben erklärt, warum wir welche Entscheidung in Bezug auf das Konzept gefällt haben.

Nach der Erläuterung des Konzepts haben wir dann die Umsetzung dieses Konzepts zu einer Implementierung beschrieben. Es wurde gezeigt, wie welche Design-Entscheidungen in der Implementierung umgesetzt wurden und welche zusätzlichen Forderungen sich dabei für die Implementierung ergeben haben.

Nach der Beschreibung der Implementierung haben wir anhand eines Beispiels gezeigt, wie der Prototyp arbeitet, wie er das Geschehen während des Simulationslaufes darstellt und wie man das Dargestellte interpretieren muss.

Wir haben Anregungen für die Weiterentwicklung des Prototyps gegeben, denen man in zukünftigen Arbeiten nachgehen kann. Diesen kann der hier entwickelte Prototyp als Grundbaustein dienen.

A Ausführung des Prototyps

A.1 Verwendete Software

Für die Implementierung des Prototyps wurde folgende Software verwendet:

- Betriebssystem: Linux (Debian GNU 3.0 bzw. SuSE 8.0)
- Gnu C++-Compiler: gcc version 3.0.4
- RTI-1.3NGv6 für RedHat7.2, i386 und gcc-3.0.2, optimiert für Multi-Threading
- GNU Make version 3.79.1

A.2 Hinweise zur Installation der RTI

Das Installationsskript der RTI ist auf die RedHat-Distribution 7.2 angepasst. Um es unter SuSE installieren zu können, ist deshalb eine Abänderung des Skriptes notwendig:

- Das Skript verwendet den Befehl *uncompress -d*. Dieser Befehl ist auf der verwendeten SuSE Distribution nicht vorhanden, kann aber problemlos durch den Befehl *compress -d* ersetzt werden, der die gleiche Funktion ausführt.

Der Patch zum Anwenden auf das Original-Installationsskript liegt dem Quellcode des Prototyps bei. Korrekte Funktionalität kann aber bei einer von der hier verwendeten Umgebung abweichenden nicht garantiert werden.

A.3 Organisation der Codeverzeichnisse

Die Verzeichnisstruktur des Quelltextes gliedert sich folgendermaßen:

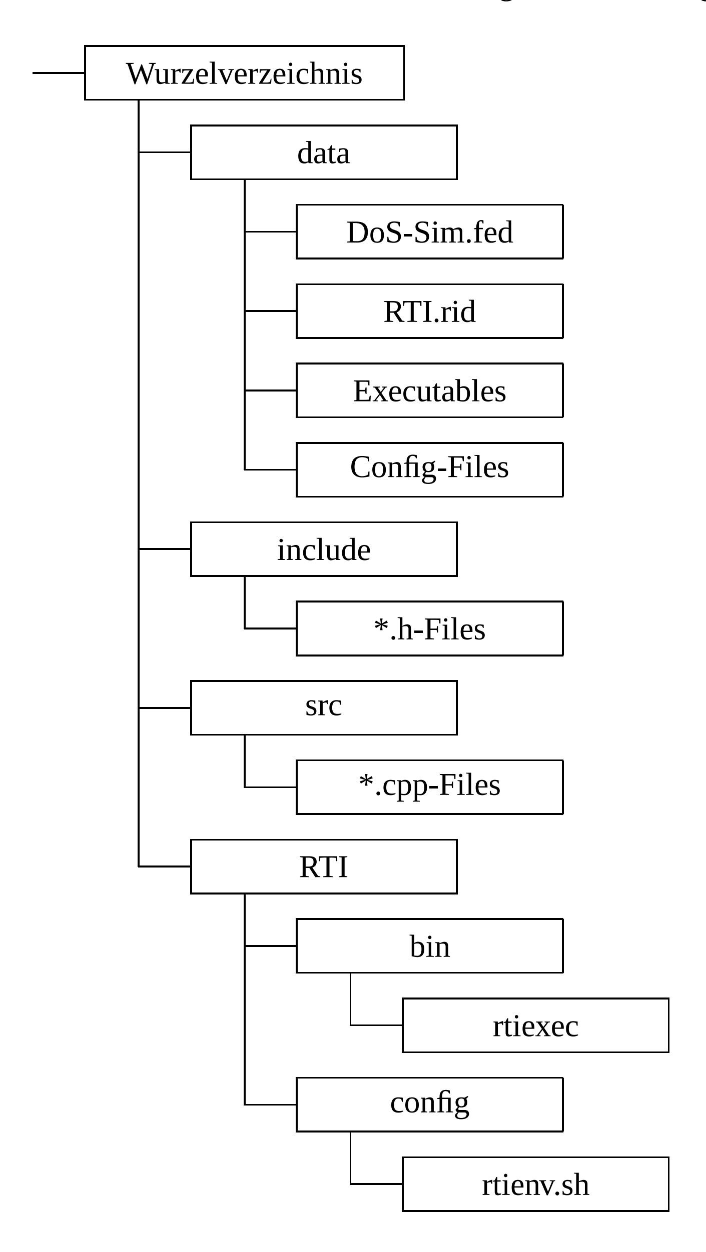


Abbildung 45: Verzeichnisstruktur

A.4 Starten der Federates

Bevor die Federates gestartet werden, sollte eine Instanz der RTI aufgerufen werden. Dazu werden zunächst mit dem Aufruf des Skripts *rtienv.sh* die nötigen Umgebungsvariablen der laufenden Shell gesetzt, dann erfolgt der Aufruf der RTI mit *rtiexec*.

Beispiel: Angenommen, man steht im Wurzelverzeichnis des Quellcodes, dann führt man folgende Befehle aus:

```
$ .  RTI/config/rtienv.sh
$ RTI/bin/rtiexec
```

Hinweis: Der Punkt vor dem Aufruf von *rtienv.sh* ist nötig, damit die Umgebungsvariablen für die aktuelle Shell gesetzt werden. Vergisst man ihn, so kann die RTI nicht ausgeführt werden.

Wenn man eine RTI am Laufen hat, so kann man die Federates starten. Dabei sollte mit dem Starten der Intranets begonnen werden, die Angreifer sind als Letztes zu starten.

Zum Start der Federates ist auch der Aufruf des Skriptes *rtienv.sh* nötig.

Ein Aufruf des Intranets sollte aus dem Verzeichnis `data` erfolgen und wird auf folgende Art durchgeführt:

```
data$ .  ../RTI/config/rtienv.sh
data$ ./intranet <NetzID> <Anz. Server> [Config]
```

Als Parameter kann man dem Intranet also seine NetzID und die Anzahl der im Intranet agierenden Server angeben. Optional kann ein Configfile für die Firewall-Regeln angegeben werden, mit dem der Router des Intranets initialisiert wird.

Der Angreifer für den nicht-verteilten SYN-Angriff wird durch diesen Aufruf gestartet:

```
data$ .  ../RTI/config/rtienv.sh
data$ ./attacker <NetzID>
```

und der Smurf-Angreifer durch folgenden:

```
data$ .  ../RTI/config/rtienv.sh
data$ ./smurf <NetzID> <Opfer-IP> <Amplifier-Netz>
```

Hier stehen <NetzID> für die Heimatnetz-ID des Angreifernetzes, <Opfer-IP> für die Adress-IP des Opfers und <Amplifier-Netz> bei dem Smurf-Angreifer für das Netzwerk, dessen Broadcast-Adresse gepingt werden soll.

Bei einem falschen Aufruf geben die Programme einen Hinweis, wie sie richtig aufzurufen sind. Hier erfolgt dann auch noch eine kurze Erklärung der Parameter.

A.5 Kompilieren der Federates

Hinweis: Das Makefile benutzt zum Aufruf des Compilers den Befehl *g++3*. Ist dieser Befehl nicht vorhanden, so muss er entweder durch den auf dem System vorhandenen Aufruf für den g++ Version3 ersetzt werden (Bei Debian GNU 3.0 stable: *g++-3.0*), oder es kann ein symbolischer Link des Namens 'g++3' angelegt werden, der auf die Binärdatei des g++-V3 zeigt.

Die Make-Aufrufe für die verschiedenen Kompilierungs-Aktionen sind aus dem `src`-Verzeichnis wie folgt aufzurufen:

- Alle Federates: *make all*
- Intranet-Federate: *make intranet*
- Angreifer-Federate: *make angreifer*
- Smurf-Angreifer: *make smurf*
- löschen aller Object-Files und Executables: *make clean*

Bei einem Makeaufruf werden alle benötigten Klassen kompiliert, soweit sie nicht schon in einer aktuellen Version vorliegen. Aber Achtung: das Makefile erkennt keine Änderungen der include-Dateien. Werden die includes verändert, so muss man die von der Änderung betroffenen Object-Files löschen und dann neu kompilieren.

B Funktionsreferenz

Dieses Kapitel ist als Referenz für jene gedacht, die sich aktiv an der Entwicklung von DoS-Simulationen auf der Basis des hier vorgestellten Prototyps beteiligen wollen. Klassenweise werden die öffentlichen Funktionen erklärt, und es wird beschrieben, für welchen Zweck sie gedacht sind.

Die privaten Funktionen werden hier weggelassen, da sie nur für interne Zwecke gedacht sind. Wer diese Abläufe verstehen will, sollte gleich den gesamten Quelltext studieren.

B.1 Klasse `IP`

`IP(int)`
: Legt eine IP mit dem übergebenen Wert an. Dieser ist der aus HostID und NetzID kombinierte Wert.

`IP(int n,int h)`
: Legt eine IP mit der NetzID n und der HostID h an. Resultierende IP ist ”n.h”

`IP(char n, int h)`
: Legt eine IP mit einer freien NetzID und der HostID h an. Resultierende IP ist ”*.h”

`IP(int n, char h)`
: Legt eine IP mit der NetzID n und einer freien HostID an. Resultierende IP ist ”n.*”

`IP(char n, char h)`
: Legt die IP ”*.*” an.

`IP(char*)`
: Erzeugt aus einem String einer IP ein Objekt IP

`const char* str()`
: Wandelt die IP in einen String um. Der Pointer auf diesen String wird zurückgegeben.

`int hostID()`
: Gibt die HostID der IP zurück.

`int netID()`
: Gibt die NetzID der IP zurück.

`int isAnyNet()`
: Gibt 1 zurück, falls die NetzID nicht auf einen festen Wert gesetzt ist, ansonsten 0.

`int isAnyHost()`
: Gibt 1 zurück,wenn die HostID frei (=”*”) ist, ansonsten 0.

`int isEqual(IP* n)`
: Vergleicht die IP mit der IP n. Sind die Werte identisch, wird 1 zurückgegeben, sonst 0.

B.2 Klasse `Net`

`Net(int)`
: Der Konstruktor der Klasse. Damit wird die Klasse gleich mit einer NetzID initialisiert.

`void newTime(RTIfedTime)`
: Sollte von der Anwendung jedesmal aufgerufen werden, wenn der Federate von der RTI einen TimeGrant erhalten hat. Durch diese Funktion wird die neue Zeit im Netzwerk zu allen Servern und dem Router propagiert.

`int getNetID()`
: Gibt die NetID zurück. Wichtig, damit der Federate-Ambassador die NetzID seines Federates bestimmen kann.

`void dumpHostlist()`
: Gibt die Liste der dem Netz bekannten Server samt ihrem Zustand auf der Standardausgabe aus.

`RTIhandler* getHandler()`
: Gibt einen Verweis auf den RTIhandler zurück. Auf diese Weise initialisieren sich die Serverobjekte ihren Zugriff auf den RTIhandler.

`void setHandler(RTIhandler*)`
: So erhält das Netz die Adresse des RTIhandlers. Der Aufruf dieser Funktion muss erfolgen, bevor das Netz arbeiten kann, und sollte daher unmittelbar nach dem Erzeugen des RTIhandler-Objektes erfolgen.

`int setRouter(Router*)`
: Diese Funktion wird von `Router` benutzt, um sich dem Netzwerk bekannt zu machen. Als Antwort wird die NetzID zurückgegeben.

`void sendPacket(Packet*)`
: Dieser Befehl reicht ein Paket an das Netzwerk weiter.

`int setServer(Server* ip, TStates s)`
: Setzt den Status des Servers ip auf den Wert s. Rückgabewert ist 0 bei Erfolg und -1, wenn ein Fehler aufgetreten ist.

`Server* addServer(int i)`
: Bringt das Netzwerk dazu, einen neuen Server mit der HostID i anzulegen. Die Adresse des neuen Serverobjektes wird dann zurückgegeben.

`Server* addServer(int i , char* f)`
: Legt einen neuen Server mit der HostID i und dem Configfile f an. Rückgabewert ist die Adresse des neuen Servers.

`int addServer(Server*)`
: Fügt einen Server dem Netzwerk hinzu, falls dieser eine IP aus dem entsprechenden Netz hat. Als Rückgabewert meldet 0 Erfolg und -1 gibt an, dass der Server nicht ins Netz hinzugefügt werden konnte.

`int delServer(int i)`
: Löscht den Server mit der HostID i aus dem Netz. Rückgabewert ist -1 bei einem aufgetretenen Fehler und 0, wenn der Server gelöscht werden konnte.

`int delServer(Server*)`
: Löscht den Server aus dem Netz, der als Argument übergeben wurde. Rückgabewert ist -1 bei einem aufgetretenen Fehler und 0, wenn der Server gelöscht werden konnte.

B.3 Klasse `Packet`

`Packet(IP* f,IP* t,Protocol prot,int fp,int tp,char* msg)`
: Erzeugt ein Paket mit Absendeadresse f, Zieladresse t, Protokoll prot, Absendeport fp und Zielport tp. Das Flags-Feld wird auf 0 initialisiert, egal welchem Protokoll das Paket angehört. Das Datenfeld erhält den Inhalt msg.

`Packet(IP*,IP*,Protocol,int,int,int flgs,char*)`
: Wirkt wie der erste Konstruktor: Erster Parameter Absende-IP, dann Ziel-IP, Protokoll, Absendeport und Zielport. Dann folgt die gewünschte Initialisierung des Flags-Felds, abschließend wieder der Inhalt des Daten-Feldes.

`int isSYN()`
: Fragt ab, ob das SYN-Bit gesetzt ist und gibt 1 (=gesetzt) oder 0(=nicht gesetzt) zurück.

`int isACK()`
: Prüft das ACK-Bit und liefert 1(=gesetzt) oder 0(=nicht gesetzt) zurück.

`int isFIN()`
: Liefert 1, wenn das FIN-Bit gesetzt ist, sonst 0

`int isRST()`
: Fragt ab, ob das RST-Bit gesetzt ist. 1=gesetzt, 0=nicht gesetzt.

`void setSYN()`
: Setzt das SYN-Bit des Pakets.

`void setACK()`
: Setzt das ACK-Bit.

`void setFIN()`
: Setzt das FIN-Bit.

`void setRST()`
: Setzt das RST-Bit.

B.4 Interface-Klasse `Program`

`void bootstate()`
: Diese Funktion setzt bei einem Reboot des Servers den Status des Programms auf den definierten Wert `STOPPED`

`void setServer(Server* host)`
: Diese Funktion ist dazu da, dem Programm zu sagen, auf welchem Server es läuft. Denn ein Programm muss auf Funktionen des Servers zurückgreifen können, um z.B. Pakete verschicken zu können.

`void setVulnerabilities(char* problem)`
: Diese Funktion fügt die Schwachstelle problem der Liste der Verwundbarkeiten des Programms hinzu.

`void patchVulnerabilities(char* problem)`
: Diese Funktion versucht, die übergebene Schwachstelle aus der Liste der Verwundbarkeiten zu löschen. Ist die Schwachstelle nicht in dem Programm, so kann sie nicht gelöscht werden, es tritt aber kein Fehler auf.

`virtual int exec(list<char*>)`
: Für die einmalige Ausführung eines Programms. Diese Funktion muss aktiv aufgerufen werden, es gibt keine Situation, in der dies automatisch geschieht.

`virtual int start(list<char*> arg)`
: Startet das Programm mit den Argumenten, die als Liste übergeben wurden. Soll das Programm je nach Funktion in den Zustand `RUNNING` oder `LISTENING` versetzen.

`virtual char* run()`
Diese Funktion wird vom Server aufgerufen, wenn dieser nach einem Time-Advance seine Arbeit durchführt und er das Programm im Zustand `RUNNING` vorfindet.

`virtual char* run(char* mesg)`
Funktion, die bei Eintreffen eines Pakets auf einem Port, auf dem das Programm läuft, aufgerufen wird.

`virtual char* stop()`
Soll die Ausführung des Programms beenden und es in den Zustand `STOPPED` versetzen.

`virtual void setVulnerabilities()`
Mit dieser Funktion sollen die Standard-Verwundbarkeiten des Programms festgelegt werden.

B.5 Klasse `Router`

`Router(Net*)`
Erzeugt einen Router, der defaultmäßig allen Paketverkehr weiterleitet.

`Router(Net*, char*)`
Erzeugt einen Router mit dem Regelsatz, der in dem File, das als zweites Argument angegeben wird, steht.

`void newTime()`
Gibt dem Router zu erkennen, dass in der Zeit vorangeschritten wurde und er die neu eingetroffenen Pakete bearbeiten muss.

`Policies getPolicy()`
Gibt die Default-Policy des Routers zurück

`int setPolicy (Policies)`
Setzt die Default-Regel auf den übergebenen Wert

`int loadRuleset(char* file)`
Lädt einen neuen Regelsatz aus dem angegebenen File ein. Der alte Regelsatz wird komplett ersetzt.

`int clearRuleset()`
Löscht alle Regeln des Regelsatzes. Es wird nun immer die Default-Policy verwendet, bis neue Regeln eingegeben werden.

`int listRuleset()`
Gibt den Regelsatz mit Nummerierung der Regeln auf die Standardausgabe.

`int dumpRuleset()`
Gibt den aktuellen Regelsatz so auf der Standardausgabe aus, wie er in einem Regelsatzfile zum Einlesen stehen müsste.

`int dumpRuleset(char* file)`
Schreibt den Regelsatz in ein File, aus dem er später wieder eingelesen werden kann.

`int mvRuleUp(int i)`
Tauscht die Position der i-ten und der $(i-1)$-ten Regel

`int mvRuleDown(int nr)`
Tauscht die Position der i-ten und der $(i+1)$-ten Regel

`int addRule(Rule* regel)`
Fügt das Objekt einer Regel an das Ende des Regelsatzes an.

`int delRule(int i)`
Löscht die Regel an i-ter Position.

`void sendPacket(Packet* packet)`
Wird benutzt, um ein Paket an den Router zu senden.

`void setHandler ( RTIhandler* )`
Initialisiert den RTIhandler, damit der Router Pakete an das Internet (RTI) schicken kann.

B.6 Klasse `RTIhandler`

`RTIhandler(RTI::RTIambassador*)`
Erzeugt einen RTIhandler mit der Adresse des RTI-Ambassadors, den der Federate benutzt.

`int tellNewInstall(IP*)`
Erzeugt eine Meldung an die RTI, dass der Server mit der angegebenen IP neu installiert wurde.

`int tellOffline(IP*)`
Meldet der RTI, dass der Server mit der übermittelten IP offline gegangen ist/ immer noch offline ist.

`int tellOnline(IP*)`
Übermittelt an die RTI, dass der Server mit der angegebenen IP online ist.

`int tellOutofOrder(IP*)`
Teilt der RTI mit, dass der Server mit der übergebenen IP im Zustand Out-of-Order ist.

`int sendPacket(Packet*)`
Sendet ein Paket an die RTI, damit es andere Netzwerke erreichen kann.

B.7 Klasse `Rule`

`Rule(IP,IP,int,int,Protocol,Policies)`
: Erzeugt ein Objekt vom Typ Rule. Die Parameter sind in folgender Reihenfolge : Absendeadresse, Zieladresse, Absendeport, Zielport, Protokoll und gewünschte Aktion.

`Policies match(IP*,IP*,int,int,Protocol)`
: Überprüft, ob die Regel mit den übergebenen Daten übereinstimmt, also zutreffend ist. Wenn dies der Fall ist, wird die in der Regel definierte Aktion zurückgegeben.

B.8 Klasse `Server`

`Server(int,Net*)`
: Konstruktor, der aus einem Integer-Wert die IP berechnet, und damit und mit der Adresse des Heimatnetzes ein Objekt `Server` erzeugt.

`Server(int,int,Net*)`
: Konstruktor, der aus der Angabe von NetzID, HostID und Adresse des Heimatnetzes ein Objekt Server erzeugt.

`Server(IP*,Net*)`
: Erzeugt ein Serverobjekt aus einer IP und der Adresse des Heimatnetzes.

`Server(int,int,char*,Net*)`
: Legt ein Objekt Server an mit NetzID, HostID, Konfigurationsfile des Servers und dem Heimatnetz.

`Server(int,char*,Net*)`
: Erzeugt ein Objekt Server aus einem Integer für die IP, dem Konfigurationsfile und dem Heimatnetz.

`Server(IP*,char*,Net*)`
: Konstruktor, der aus einem IP-Objekt, einem Konfigurationsfile und der Adresse des Heimatnetzes ein Serverobjekt erzeugt.

`void newTime(RTIfedTime)`
: Funktion, mit der der Time-Advance an den Server weitergegeben wird. Der Server wird hier sein Tcon-Objekt updaten und seine Paket-Queues umsortieren, um in der neuen Zeit richtig arbeiten zu können. Hier wird auch geprüft, ob der Server wieder automatisch booten soll (z.B. nach einer Neuinstallation).

`int doWork()`
Die Arbeit des Servers zur aktuellen Zeit wird hiermit ausgeführt. Damit werden die laufenden Programme für ihre Aktionen angestoßen und die erhaltenen IP-Pakete werden bearbeitet.

`int setConfig(char*)`
Funktion, mit der man dem Server ein neues Konfigurationsfile zuweisen kann.

`int boot()`
Der Server wird gebootet. Dabei werden alle Programme, bei denen die Variable `startAtBoot` gesetzt ist, gestartet. Der Rechner erhält den Status `ONLINE`.

`int shutDown()`
Der Server wechselt in den Zustand `OFFLINE`. Hierbei werden alle Programme beendet, die TCP/UDP-Ports werden freigegeben.

`int newInstallation()`
Zuerst wird der Server heruntergefahren, wechselt damit in den Zustand `OFFLINE`. Dann werden seine Programme gelöscht. Nun werden die im Konfigurationsfile angegebenen Programme installiert. Die Zeit bis zum automatischen Boot wird auf den durch die Konstante `DOWNTIME` (in `server.h`) definierten Wert gesetzt.

`int newInstallation(char* file)`
Siehe oben, aber hier wird gleichzeitig das Konfigurationsfile neu gesetzt.

`void send(Packet*)`
Funktion, um ein Paket an den Server zu senden.

`int sendTo(int,IP*,int,Protocol,char*)`
Funktion, mit der Programme, die auf dem Rechner installiert sind, Pakete an andere Rechner verschicken können. Es erfolgt die Angabe des Absende-Ports, der Ziel-IP, des Zielports, des Protokolls und des Inhalts des Paketes. Da keine Absendeadresse angegeben wird, wird automatisch die korrekte IP des Rechners als Absende-IP verwendet. Selbst wenn das Datenfeld leer sein sollte, wird hier eine Verbindung mit Datenaustausch angenommen. Alternativ kann sonst die Funktion `connect` verwendet werden.

`int sendTo(IP* fip,int fport,IP*,int,Protocol,char*)`
Wie zuvor, aber durch Angabe einer expliziten Absende-IP (erster Parameter) kann so ein Paket mit gespooftem Absender erzeugt werden.

`int connect(int fport,IP* ip,int tport )`
Mit dieser Funktion kann ein Connect auf einen Port eines anderen Rechners gemacht werden. Bei einem Connect ist keine falsche Absendeadresse

möglich. Dazu könnte aber `sendTo()` mit Angabe einer falschen Adresse benutzt werden.

`int getHostID()`
: Liefert dem Aufrufer der Funktion die HostID des Rechners zurück.

`int getNetID()`
: Gibt dem aufrufenden Programm die NetzID des Servers zurück.

`int install( Program*)`
: Installiert das übergebene Programmobjekt als Programm auf dem Server.

`int install(std::string)`
: Installiert ein Programm von Typ des übergebenen Strings auf dem Server. Als Programmname wird hierbei der Klassenname des Programmes gewählt.

`int install(std::string,char*)`
: Installiert ein Programm von Typ des angegebenen Strings auf dem Server. Das Programm erhält hierbei als Name den als 2. Argument übergebenen String.

`int deinstall(char*)`
: Deinstalliert das Programm, dessen Name als Argument angegeben wurde, falls ein solches existiert.

`int adduser(const char*)`
: Legt einen Benutzer mit dem Argument als Passwort an. Rückgabewert ist die UserID des neuen Benutzers.

`int deluser(int)`
: Löscht den User mit der angegebenen ID.

`int registerPort(Program*,int port,Protocol)`
: Versucht, das Programm, welches beim Aufruf übergeben wurde, auf dem angegebenen Port des Protokolls zu registrieren. Schlägt dieser Versuch fehl, so wird ein negativer Wert zurückgegeben (-2 für Port out of range, -1 für Port schon belegt), bei Erfolg ist der Rückgabewert 0.

`int unregisterPort(Program*,int port,Protocol)`
: Diese Funktion dient dazu, dass ein Programm bei seiner Beendigung den von ihm belegten Port wieder freigeben kann. Rückgabewerte sind -2 für Port außerhalb der Portrange, -1, wenn das Programm nicht auf dem Port läuft, und 0 bei erfolgter Portfreigabe.

`int chpasswd(int,char*,char*)`
: Diese Funktion dient dazu, einem User das Wechseln des Passworts zu erlauben. Hierzu werden UserID, altes Passwort und neues Passwort als Argumente benötigt. Der Wechsel ist nur dann erfolgreich, wenn das alte Passwort mit dem in der Serverliste angegebenen Passwort des Users übereinstimmt.

int execBefehl(char* n ,list<char*>)
: Führt einen Befehl namens n aus (mittels Programm::exec()) und übergibt diesem die angegebene Argumentliste.

int execBefehl(int id ,char*,list<char*>)
: Führt den als zweites Argument angegebenen Befehl als User id aus und übergibt ihm die Argumentliste.

int startService(char*)
: Startet einen Service (Programm::start()) des angegebenen Namens mit leerer Argumentliste.

int startService(char*,list<char*>)
: Startet einen Service des angegebenen Namens mit der angegebenen Argumentliste.

int startService(int,char*)
: Startet als User mit der ID des ersten Arguments den Service mit dem Namen aus Argument 2.

int startService(int,char*,list<char*>)
: Der User mit der ID aus Argument 1 startet das Programm mit dem Namen aus Argument 2 und der Parameterliste aus Argument 3.

int stopService(char*)
: Ruft Programm::stop() des Programms auf, das unter dem Namen, der in Argument 1 angegeben wurde, läuft.

int login(int,char*)
: Versucht, den User mit der ID aus Argument 1 mit dem Passwort aus Argument 2 auf dem System einzuloggen.

int logout(int)
: Meldet den User mit der angegebenen ID vom System ab.

B.9 Klasse Tcon

Tcon()
: Erzeugt ein Objekt vom Typ Tcon. Dieses Objekt ist für die Verwaltung der eingehenden Verbindungen zuständig, es merkt sich, wieviele nicht vollständig aufgebaute Verbindungen zu welchem Zeitpunkt wieder frei werden und ob noch Verbindungen angenommen werden können.

void reset()
: Setzt die Anzahl der belegten Verbindungen auf 0, um z.B. bei einem Reboot nicht noch alte Verbindungen zu übernehmen.

`int getAnz()`
: Gibt die Anzahl der belegten Verbindungsressourcen an.

`int inc()`
: Fügt einen neuen Verbindungsaufbauversuch hinzu.

`int con_possible()`
: Liefert true (=1) zurück, falls noch Verbindungen aufgebaut werden können, sonst false (=0).

`void update(RTIfedTime time)`
: Funktion, die nach jedem TimeAdvance aufgerufen werden muss. Sie löscht die Verbindungsversuche, die in einen TimeOut laufen, und gibt damit ihre Ressourcen wieder frei.

B.10 Programm-Klassen

Bei den Programmklassen werden nur die neu implementierten Funktionen beschrieben. Die Funktionen, die schon in `Program` behandelt wurden, werden nicht noch einmal beschrieben. Alle in einer Programm-Klasse nicht erwähnten Funktionen, die in `Program` als virtual deklariert sind, haben keine Funktion. Sie geben im Allgemeinen eine Fehlermeldung aus, wenn sie aufgerufen werden.

B.10.1 Programm `IDT`

`IDT(char*)`
: Konstruktor, der ein Objekt vom Typ `IDT` mit einem speziellen Namen erzeugt. Der Server wird nicht gesetzt, dies passiert bei Installation des Programms auf dem Server durch Aufruf von `setServer()`. Die Referenzliste des IDTs ist leer.

`IDT(char*,Server*)`
: Ordnet dem `IDT`-Objekt neben einem Namen noch einen Server zu. Dies ist der Server, den das Programm als seinen Host-Rechner betrachtet, also als den Rechner, auf dem es läuft. Die Referenzliste der Programme ist leer.

`int exec(list<char*>)`
: Wird als Parameter eine einelementige Liste mit dem Inhalt „update” übergeben, so legt das Programm eine neue Referenzliste mit den auf dem Host-Rechner installierten Programmen an. Diese Funktion wird als Update-Funktion bezeichnet.

int start(list<char*>)
: Startet das IDT. Sollte die Referenz-Liste der auf dem Server installierten Programme leer sein, so wird sie neu angelegt, sonst wird die alte Liste weiterverwendet. Der Zustand des Programms wird nach RUNNING gesetzt.

int run()
: Ruft das IDT auf, damit dieses die Referenzliste der Programme mit der aktuellen Installation vergleicht. Ist sie gleich, so passiert nichts, ist sie verändert, so sorgt das IDT für eine Neuinstallation des Servers.

int stop()
: Stoppt das IDT, dieses wechselt in den Zustand STOPPED. Die Programm-Referenzliste bleibt bestehen.

B.10.2 Programm Ping

Ping(char*)
: Konstruktor, der dem Ping-Programm einen speziellen Namen zuordnet. Der Server wird nicht gesetzt, dies passiert bei Installation des Programms auf dem Server durch Aufruf von setServer().

Ping(char*,Server*)
: Ordnet dem Programm neben einem Namen noch einen Server zu. Dies ist der Server, den das Programm als seinen Host-Rechner betrachtet, also als den Rechner, auf dem es läuft.

int exec(list<char*>)
: Das Programm wird genau ein ICMP-Ping-request-Paket an jeden Server schicken, der in der Argumentliste steht. Die IP dieser Server hat als C-String in der Liste zu stehen.

B.10.3 Programm Smurf

Smurf(char*)
: Konstruktor, der dem Smurf-Programm einen speziellen Namen zuordnet. Der Server wird nicht gesetzt, dies passiert bei Installation des Programms auf dem Server durch Aufruf von setServer().

Smurf(char*,Server*)
: Ordnet dem Programm neben einem Namen noch einen Server zu. Dies ist der Server, den das Programm als seinen Host-Rechner betrachtet.

int exec(list<char*>)
: Sendet genau einen Zeitschritt lang 10 Pakete an die Adresse, die an zweiter Position in der Argumentliste steht. Das erste Listenelement gibt die Absender-IP (als C-String) an, die beim Versenden der Pakete angegeben werden soll. Weitere Listenelemente werden ignoriert. Sind weniger als zwei Elemente in der Liste, bricht das Programm die Ausführung ab. Zurückgegeben wird dann der Wert -1, bei korrekter Ausführung 1.

int start(list<char*>)
: Bei dieser Funktion wird dasselbe ausgeführt wie bei exec(), aber zusätzlich wird der Programm-Zustand auf RUNNING gesetzt. Die übergebenen Argumente werden in eine Liste übertragen, auf die das Programm auch in den folgenden Zeittakten Zugriff hat. Der Schleifenzähler wird auf einen Startwert gesetzt.

char* run()
: Das Programm führt einmal das aus, was in exec() ausgeführt wird, benutzt aber die Argumente aus der dauerhaften Liste, die durch start() ihre Werte gesetzt bekommen hat. Der Schleifenzähler wird dekrementiert. Wird er 0, geht das Programm in den Zustand STOPPED über.

int stop()
: Setzt den Zustand des Programms auf STOPPED. Weiter wird die dauerhafte Liste gelöscht, in der die Argumente für den Angriff stehen.

B.10.4 Programm SynAttackTool

SynAttackTool(char*)
: Konstruktor, der ein SynAttackTool-Objekt mit einem speziellen Namen erzeugt. Der Server wird nicht gesetzt.

SynAttackTool(char*,Server*)
: Erzeugt ein Programm-Objekt, das einen Namen und einen Server zugewiesen bekommt.

int exec(list<char*>)
: Startet eine SYN-Attacke, die über 10 Zeitschritte läuft. Die SYN-Attacke versucht, den Port 8 auf Opferseite für die Attacke zu nutzen. Dieses ist der Port, auf dem im Normalfall Programme von Typ Webserver laufen. exec() setzt den Status des Programms auf RUNNING, damit das Programm auch in den folgenden Zeitschritten ausgeführt wird. Ein Zähler *turns* wird auf den Wert gesetzt, wie lange das Programm laufen soll (hier 10).

`char* run()`
: Führt das Programm weiter aus, dekrementiert hierbei in jedem Zeitschritt den Zähler *turns*. Ist dieser bei 0 angekommen, geht das Programm in den Zustand `STOPPED` über.

`int stop()`
: Bricht die `exec()`-Ausführung ab, setzt *turns* auf 0 und setzt den Programmzustand auf `STOPPED`.

B.10.5 Programm `Webserver`

`Webserver(char*)`
: Konstruktor, der ein `Webserver`-Objekt mit einem speziellen Namen erzeugt. Der Server wird nicht gesetzt.

`Webserver(char*,Server*)`
: Konstruktor, der ein `Webserver`-Objekt mit einem speziellen Namen erzeugt und dabei auch den Host-Server setzt.

`int start(list<char*>)`
: Startet den Webserver, der damit in den Zustand `LISTENING` übergeht. Das Programm registriert sich bei dem ihm zugewiesenen Port (standardmäßig Port 8, kann aber mit `setPort()` verändert werden).

`int run(char*)`
: Wird aufgerufen, wenn ein Paket auf dem Port des Webservers eintrifft. Der übergebene Parameter ist der Datenteil des Pakets.

`int stop()`
: Stoppt den Webserver, der sich damit vom Port löst und in den Zustand `STOPPED` übergeht.

`int setPort()`
: Funktion, um den Port zu verändern, auf dem sich der Webserver registriert. Standardport ist 8.

Literatur

[1] M. Allman and S. Ostermann. One: The Ohio Network Emulator, 1997.

[2] Jason Barlow and Woody Thrower. TFN2K - An Analysis, 2000.
`http://packetstormsecurity.nl/distributed/TFN2k_Analysis-1.3.txt`.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *Das UML Benutzerhandbuch*. Addison Wesley, 1999.

[4] Russell Bradford, Rob Simmonds, and Brian Unger. A Parallel Discrete Event IP Network Emulator. In *MASCOTS*, pages 315–, 2000.

[5] Lawrence S. Brakmo and Larry L. Peterson. Experiences with Network Simulation. In *Measurement and Modeling of Computer Systems*, pages 80–90, 1996.

[6] Christoph Busch and Stephen Wolthusen. *Netzwerksicherheit*. Spektrum Akademischer Verlag, 2002.

[7] Laurent Cazard and Martin Adelantado. HLA Federates Design And Federations Management: Towards a Higher Level Object-Oriented Architecture Hiding The HLA Services, 2002.

[8] Cisco. Defining Strategies to Protect Against TCP SYN Denial of Service Attacks, 2003.
`http://cio.cisco.com/warp/public/707/4.html`.

[9] CERT coordination Center. Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks, 2000.
`http://www.cert.org/advisories/CA-1998-01.html`.

[10] CERT coordination Center. Trends in Denial of Service Attack Technology, October 2001. v1.0,
`http://www.cert.org/archive/pdf/DoS_trends.pdf`.

[11] David Dittrich. The DoS Project’s ”trinoo” distributed denial of service attack tool, 1999.
`http://staff.washington.edu/dittrich/misc/trinoo.analysis.txt`.

[12] David Dittrich. The ”stacheldraht” distributed denial of service attack tool, 1999.
http://staff.washington.edu/dittrich/misc/stacheldraht.analysis.txt.

[13] David Dittrich. The ”tribe flood network” distributed denial of service attack tool, 1999.
http://staff.washington.edu/dittrich/misc/tfn.analysis.txt.

[14] David Dittrich, Sven Dietrich, and Neil Long. Analysing Distributed Denial of Service Attack Tools: The Shaft Case. In *Proceedings of the 14th Systems Administration Conference (LISA2000)*, 2000.

[15] David Dittrich, George Weaver, Sven Dietrich, and Neil Long. The ”mstream” distributed denial of service attack tool, 2000.
http://staff.washington.edu/dittrich/misc/mstream.analysis.txt.

[16] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Oldenbourg Wissenschaftsverlag GmbH, 2001.

[17] P. Ferguson. Defeating Denial of Service Attacks which employ IP Source Address Spoofing, 1998. rfc2267.

[18] S. Floyd and V. Paxson. Difficulties in Simulating the Internet, 2001.

[19] Thomer M. Gil and Massimiliano Poletto. MULTOPS: a Data-Structure for Bandwidth Attack Detection. In *Proceedings of the 10th USENIX Security Symposium*, pages 23–38, 2001.

[20] Axel Hagedorn. Distributed Denial of Service: Angriffswerkzeuge und Abwehrmöglichkeiten, 2003. Seminar: Ausgewählte Aspekte zum Thema IT-Sicherheit, Fachgebiet Sicherheit in der Informationstechnik, TU Darmstadt.

[21] Craig A. Huegen. The latest in Denial of Service Attacks: ”Smurfing” - Description and Information to minimize Effects, 2000.
http://www.pentics.net/denial-of-service/white-papers/smurf.cgi.

[22] SANS Institute. Consensus Roadmap for Defeating Distributed Denial of Service Attacks, 2000. Version 1.10,
http://www.sans.org/dosstep/roadmap.php.

[23] SANS Institute. Help Defeat Denial of Service Attacks: Step-by-Step, 2000. Revision 1.4,
http://www.sans.org/dosstep/index.php.

[24] John Ioannidis and Steven M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Proceedings of Network and Distributed System Security Symposium, Catamaran Resort Hotel San Diego, California 6-8 February 2002*, 1775 Wiehle Ave., Suite 102, Reston, VA 20190, Februar 2002. The Internet Society.

[25] ju@onlinesicherheit.de. Internet Angriffe mit DDoS und DoS Strategien, 2000.
http://www.computec.ch/dokumente/denial_of_service/internet_angriffe_mit_ddos_und_dos/internet_angriffe_mit_ddos_und_dos.txt.

[26] Stephan Kallnik, Daniel Pape, Daniel Schröter, and Stefan Strobel. Das Sicherheitsloch - Buffer-Overflows und wie man sich davor schützt. *c't*, 23/2001:216, 2001.
http://www.heise.de/ct/01/23/216/.

[27] HyungJong Kim, KyungHee Koh, DongHoon Shin, and HongGeun Kim. Vulnerability Assessment Simulation for Information Infrastructure Protection. In *Infrastructure Security, International Conference, InfraSec 2002 Bristol, UK, October 1-3, 2002, Proceedings*, pages 145–161, 2002.
http://www.informatik.uni-trier.de/~ley/db/conf/infrasec/infrasec2002.html.

[28] Mike Kristovich. Topic: Multi-vendor Game Server DDoS Vulnerability, 2002. Security Advisory MK#001, Pivx Solutions,
http://www.pivx.com/kristovich/adv/mk001/.

[29] Dr. F. Kuhl, Dr. R. Weatherly, and Dr. J. Dahmann. *Creating Computer Simulation Systems - An Introduction To The High Level Architecture*. Prentice Hall, 1999.

[30] Ping-Herng Denny Lin. Survey of Denial of Service Countermeasures, 2000.
http://www.lasierra.edu/~dlin/classes/cpsc433/cpsc433.htm.

[31] Jelena Mirkovic, Janice Martin, and Peter Reiher. A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms. Technical Report 020018, Computer Science Department, University of California, Los Angeles, 2002.

[32] Jelena Mirković, Gregory Prier, and Peter Reiher. Attacking DDoS at the Source. In *Proceedings of ICNP 2002*, pages 312–321, November 2002.

[33] Ingo Molnar. [announcement] ”Exec Shield”, new Linux security feature, 2003.
http://people.redhat.com/mingo/exec-shield/ANNOUNCE-exec-shield.

[34] John R. Mostow, John D. Roberts, and John Bott. Integration of an Internet Attack Simulator in an HLA Environment, 2000.

[35] NN. TCP/IP-Angriffe, 1999.
http://www.computec.ch/dokumente/denial_of_service/tcp-ip-angriffe/tcp-ip-angriffe.html.

[36] Vern Paxson. An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks. *ACM Computer Communications Review (CCR)*, 31(July), 2001.
http://www.icir.org/vern/papers/reflectors.CCR.01.ps.gz.

[37] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Defending Against Distributed Denial of Service Attacks Using Selective Pushback, 2002.

[38] Marc Ruef. DoS (Denial of Service), 2000.
http://www.computec.ch/dokumente/denial_of_service/denial_of_service/denial_of_service.html.

[39] J. Seemann and J. W. von Gudenberg. *Software-Entwurf mit UML*. Springer, 2000.

[40] Rob Simmonds, Russell Bradford, and Brian Unger. Applying parallel discrete event simulation to network emulation. In *Workshop on Parallel and Distributed Simulation*, pages 15–22, 2000.

[41] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison Wesley, 2000.

[42] Heise Zeitschriften Verlag. Denial of Service bei Freemailer GMX, 2000. heise online news vom 15.12.2000,
http://www.heise.de/newsticker/data/hob-15.12.00-000/.

[43] Heise Zeitschriften Verlag. DoS-Attacke auf Darmstädter Unternehmen, 2000. heise online news vom 19.12.2000,
http://www.heise.de/newsticker/data/hob-19.12.00-000/.

[44] Heise Zeitschriften Verlag. DoS-Attacke gegen Websites von Microsoft, 2001. heise online news vom 26.01.2001, `http://www.heise.de/newsticker/data/jk-26.01.01-000/`.

[45] Heise Zeitschriften Verlag. Internet-Wurm startet DoS-Attacke, 2001. heise online news vom 07.06.2001, `http://www.heise.de/newsticker/data/fr-07.06.01-000/`.

[46] Heise Zeitschriften Verlag. Unruhe im DNS: Alle Maschinen stop, Gentlemen!, 2002. heise online news vom 15.08.2002, `http://www.heise.de/newsticker/data/jk-15.08.02-003/`.

[47] Heise Zeitschriften Verlag. DoS-Angriffe über Game-Server gefährden Internet-Nutzer, 2003. heise online news vom 18.01.2003, `http://www.heise.de/newsticker/data/ju-18.01.03-001/`.

[48] Heise Zeitschriften Verlag. Linux: Keine Chance für Buffer Overflows, 2003. heise online news vom 4.05.2003, `http://www.heise.de/newsticker/data/cgl-04.05.03-002/`.

[49] Heise Zeitschriften Verlag. Trusted Debian mit verschärftem Sicherheitskonzept für Linux, 2003. heise online news vom 22.04.2003, `http://www.heise.de/newsticker/data/ola-22.04.03-002/`.

[50] Marco Vogel. Distributed Denial-of-Service (DoS): Vorgehen, Gegenmaßnahmen, 2000. Seminar Datenverarbeitung SS2000: Verteilte Systeme / Sicherheit im Internet, Ruhr-Universität Bochum, `http://www.etdv.ruhr-uni-bochum.de/dv/lehre/seminar/syssec-apr/`.

[51] Tobias Winkelmann. Systemsicherheit - Angriffspunkte eines Rechners, 2000. Seminar Datenverarbeitung SS2000: Verteilte Systeme / Sicherheit im Internet, Ruhr-Universität Bochum, `http://www.etdv.ruhr-uni-bochum.de/dv/lehre/seminar/syssec-apr/`.